Quicksort

Module 3: Efficient Sorting and Selection

Overview

Today we discuss a new sorting algorithm, *Quicksort*, and its randomized version, *Randomized-Quicksort*. *Randomized-Quicksort* is considered the fastest general-purpose sort in practice.

Goals:

- Understand Quicksort and it's best-case, worst-case and averge case complexity
- Understand Randomized Quicksort and its expected complexity

1 Quicksort overview

- Quicksort is similar to Mergesort
- Remember that Mergesort is based on the divide-and-conquer technique
  - Partition n elements array A into two subarrays of $n/2$ elements each
  - Sort the two subarrays recursively
  - Merge the two subarrays

  Mergesort running time: $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$.
- Note that in Mergesort dividing the array is easy (basically there's nothing to do beyond computing the middle index), and the work for Mergesort happens in the "merge" phase.
- Could we do it the other way around, that is, divide the elements such that there is no need of merging?
  - We would need to **partition** A into subarrays, $A = [A'A"]$, such that all elements in A' are smaller than all elements in A''
  - Recursively sort A' and $A"$
  - Once A' and $A"$ are sorted, then $A = [A'A"]$ is also sorted !

  Here the work happens in the "divide" phase, and there is no need to combine/merge because of the way A was partitioned!
- Let's think about how we might come up with such a partition procedure that does what we need

2 How to Partition?

- Here's a first idea for partitioning an array: pick an element in the array (we'll call this the pivot), and then traverse the array and put all elements $\leq$ the pivot in an array A' and all the elements $>$ pivot in an array A".

  Pretty easy, right? With our algorithms hat we ask: Can this be improved?

- In terms of time, this takes linear time, and cannot do better than this (why?); in terms of space, it uses $\Theta(n)$ additional space for A' and A".

  Which leads to the question: Could we do the Partition in-place? If we are able to come up with an in-place partition, this immediately leads to in-place quicksort. Remember that in the sorting world, in-placeness is a nice property to have.

- Take a few minutes to think about how an in-place partition might look iike. You'd pick a pivot and you'd need to move elements around, trying to get all elements smaller than the pivot to the left side of the array, and all elements larger than the pivot to the right side of the array, and then you'd put the pivot itself in between these two regions. Note that you don't know how many elements are <pivot and how many are larger —- that depends on what the pivot is.

- There are couple of ways to go about this. One way, which was described by Sir Hoare, the inventor of Quicksort, traverses the array left to right and right to left, finds an element on the left that's $>$ pivot and one on the right that's $<$ pivot, swaps them and continues.

- We'll go over a different (perhaps simpler?) way to partition, which is called LOMUTO partition, named after the researcher who first described it.

- Note: As usual with recursive functions on arrays, the indices p and r represent the part of the array that is being recursed on.

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
FOR j = p TO r − 1 DO
    IF A[j] ≤ x THEN
        i = i + 1
        Exchange A[i] and A[j]
Exchange A[i + 1] and A[r]
RETURN i + 1
```

- The idea: Partition selects the last elements as the pivot and stores it as x. Then proceeds to run a loop through all elements up to the last one (excluding the last one), while growing a (contiguous) region of elements $A[p]$ through $A[i]$ that are all $\leq$ the pivot; variable i represents the index of the last element in this region; initially the region is empty and i is set to $p-1$. Then while iterating through all elements, if the current element $A[j]$ is smaller or equal to the pivot, it means it must be moved to this region, so i is incremented and $A[i]$ is swapped

with $A[j]$. Otherwise, if it is larger than the pivot, it can stay where it is and no need to do anything.

- Example:

|	2	8	7	1	3	5	6	4	i=0, j=1
2	|	8	7	1	3	5	6	4	i=1, j=2
2	|	8	|	7	1	3	5	6	4	i=1, j=3
2	|	8	7	|	1	3	5	6	4	i=1, j=4
2	1	|	7	8	|	3	5	6	4	i=2, j=5
2	1	3	|	8	7	|	5	6	4	i=3, j=6
2	1	3	|	8	7	5	|	6	4	i=3, j=7
2	1	3	|	8	7	5	6	|	4	i=3, j=8
2	1	3	|	4	|	7	5	6	8	q=4

- Return value: After the loop is done, PARTITION swaps the pivot with $A[i+1]$ and returns $i+1$. Let's call the return value of Partition by q. All elements to the left of q are $\leq A[q]$ and all elements to the right of q are $> A[q]$

- **Why does this work?** Understanding how/why Partition works (correctly) relies on understanding the followig "invariants" (ie facts that are true) that are maintained by the algorithm. Consider the jth iteration of the loop. The following are true before this iteration starts:

  - All elements from p to i are smaller than the pivot x; mathematically this is written as: $A[k] \leq x$ for all k s.th. $p \leq k \leq i$
  - All elements from $i+1$ through $j-1$ are larger than the pivot x; mathematically this is written as: $A[k] > x$ for all k s.th. $i+1 \leq k \leq j-1$

  Note that these invariants are trivially true before the first iteration of the loop when $i = p-1, j = p$.

  The j'th iteration compares $A[j]$ with x; if $A[j] > x$ then $A[j]$ can stay where it is, so nothing needs to be done. If $A[j] \leq x$ then essentially it swaps $A[j]$ with $A[i+1]$; note that $A[i+1]$ is known to be larger than the pivot, so it's ok for it to be moved at index j; i is incremented to mark the new element smaller than the pivot that was added to the region. Thus the invariants are maintained by the jth iteration.

- **Partition analysis**: PARTITION does one pass through the input and thus runs in linear time in terms of the number of elements that it's partitioning. That's $\Theta(r-p)$.

- Once you understand it, you will probably agree that it's a nifty piece of code to be able to do everything in place and in one loop!

Self-study exercise:

- Consider the array $A = [8, 3, 1, 5, 7, 2, 9, 4]$ and call Partition on it: $Partition(A, 0, 7)$. What does the array look like after partition is called, and what is the value q returned?

3 Quicksort

- Now that we figured out how to partition an array , let's go back to Quicksort
- As usual with in-place sorting, we expect that the call to *Quicksort(A,p,r)* will sort $A[p...r]$. To sort the entire array, one would need to call QUICKSORT$(A, 0, n-1)$.
- We assume that *Partition(A, p, r)* partitions $A[p...r]$ and returns an index q (with $p \leq q \leq r$) such that $A[i] \leq A[q]$ for all $p \leq i < q$, and $A[q] \leq A[i]$ for all $q < i \leq r$
- We can write *Quicksort(A,p,r)* as follows:

```
QUICKSORT(A, p, r)
IF p < r THEN
    q=PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

- **Why does this sort (correctly)?** Let's assume that PARTITION works correctly.

  1. Let's start by arguing that Quicksort sorts correctly when $n = 1$, that is, when there's only one element in the array. In this case the indices p and r would be equal, and Quicksort would return right away.
  2. Now let's consider an array of size $n = 2$: Partition will return either $q = 0$ or $q = 1$; thus there would be one recursive Quicksort call to an array of size 1. We know those work correctly (above).
  3. Now if $n = 3$: Partition will return $q = 0$ or $q = 1$ or $q = 3$ and will give a partition of $0 - 2$ or $1 - 1$ or $2 - 0$, which will generate recursive calls to quicksort arrays of size 1 or 2 — and we know that those sort correctly from above.
  4. And so on. Basically Quicksort-ing n elements is correct assuming Partition is correct, and assuming call to Quicksort with fewer than n elemenst work correctly.

  This is the informal, bottom-up argument, but it captures the spirit. The formal way to prove this is induction. Basically, if Partition works correctly and Quicksort sorts correctly the two sides of the partition, then we can show that the whole array is sorted correctly. If you know induction, this should be easy. If you don't, no worries. The goal of this exercise is not to teach you induction, but to trigger reflection on why Quicksort sorts correctly.

4 QUICKSORT analysis

- We analyze running time of recursive algorithms by writing recurrence relations.
- Note that we don't know the index q ahead of time, because it depends on the rank of the element chosen as pivot relative to the other elements in the array

- **Self-study exercise:** Consider the array $A = [8, 3, 5, 7, 2, 9, 4, 1]$ and call Partition on it: $Partition(A, 0, 7)$. What does the array look like after partition is called, and what is the value q returned?

- **Self-study exercise:** Consider the array $A = [8, 3, 5, 7, 2, 1, 4, 9]$ and call Partition on it: $Partition(A, 0, 7)$. What does the array look like after partition is called, and what is the value q returned?

- What can we expect in terms of Quicksort running time? Let's consider some cases.

- $\frac{1}{2}$**-to-**$\frac{1}{2}$ **split:** If the index q returned by Partition is in the middle of the range $p..r$ for all recursive calls, and assuming Partition runs in $\Theta(n)$ time, we get the recurrence $T(n) = 2T(n/2) + \Theta(n)$, which is the same as Mergesort $\Rightarrow T(n) = \Theta(n \log n)$

  Best case: Perfectly balanced Partition $\Rightarrow T(n) = \Theta(n \log n)$

- $0 - (n-1)$: If Partition always returns $q = p$ or $q = r$ the running time becomes $T(n) = \Theta(n) + T(0) + T(n-1)$ which solves to $\Rightarrow T(n) = \Theta(n^2)$.

  Worst-case: all elements on one side of the pivot $\Rightarrow T(n) = \Theta(n^2)$

  Question: How does the array look to trigger worst-case for Quicksort?

  Answer: see appendix

- Let's dig a step further. At one end we have all-perfect -splits give us which gives us $\Theta(n \lg n)$; at the other end we have all-bad-splits which gives us $\Theta(n^2)$. What about ...in between? Let's consider more cases.

- $\frac{1}{3}$**-to-**$\frac{2}{3}$ **split:** Consider the case when the index returned by Partitoon leads to a $\frac{1}{3} - to - \frac{2}{3}$ split, for all recursive calls. $\Rightarrow T(n) = \Theta(n) + T(\frac{1}{3}n) + T(\frac{2}{3}n)$ which solves to $T(n) = \Theta(n \lg n)$

- $\frac{1}{4}$**-to-**$\frac{3}{4}$ **split:** Consider the case when the index returned by Partitoon leads to a $\frac{1}{4} - to - \frac{3}{4}$ split, for all recursive calls. $\Rightarrow T(n) = \Theta(n) + T(\frac{1}{4}n) + T(\frac{3}{4}n)$ which solves to $T(n) = \Theta(n \lg n)$

- $\frac{1}{10}$**-to-**$\frac{9}{10}$ **split:** Consider the case when the index returned by Partitoon leads to a $\frac{1}{10}$-to-$\frac{9}{10}$ split, for all recursive calls. $\Rightarrow T(n) = \Theta(n) + T(\frac{1}{10}n) + T(\frac{9}{10}n)$ which solves to $T(n) = \Theta(n \lg n)$

- So splits like $\frac{1}{2}$-to-$\frac{1}{2}$, $\frac{1}{3}$-to-$\frac{2}{3}$, ,,, $\frac{1}{9}$-to-$\frac{8}{9}$,.... are all good. There are a LOT of good splits!

4.1 Quicksort average-case running time

- So Quicksort takes $\Theta(n \lg n)$ in the best case and $\Theta(n^2)$ in the worst-case. We have seen, however, that there are A LOT of splits that result in best-case behavior, so it is natural to ask: what is the **average-case running time** of Quicksort? Is it closer to $\Theta(n \lg n)$ or to $\Theta(n^2)$?

- When analyzing "average" time we have to be careful with the set of inputs on which we take the average:

  - If we run QUICKSORT on a set of inputs that are all almost sorted, the average running time will be close to the worst-case.

– Similarly, if we run QUICKSORT on a set of inputs that give good splits, the average running time will be close to the best-case.

- Most of the time, average-case is analyzed assuming a *uniform distribution of the input*. This assumption has to be made explicit, but often it's missing (If anyone asks "what's the average case of algorithm X?" you should say: "What sort of input distribution can I assume?")

- What happens if we run QUICKSORT on a set of inputs which are picked uniformly at random from the space of all possible input permutations?

  Answer: The average case will be close to the best-case. Why? Intuitively, if any input ordering is equally likely, then we expect at least as many good splits as bad splits, therefore on the average a bad split will be followed by a good split, and it gets "absorbed" in the good split. To see this, consider an $\frac{n}{2}$-to-$\frac{n}{2}$ split, followed by a bad splits in both branches: 0-to($\frac{n}{2}-1$). Thus a call to Quicksort on an array of n elements generates two recursive calls on arrays of size $n/2$, and at the next level this generates two recursive calls on arrays of size $n/2-1$. This means that it takes two levels of recursion to get from a problem of size n to two problems of size $n/2-1$. This means that on the average the recursion depth is $2\lg n$ instead of $\lg n$.

  So, under the assumption that **all input permutations are equally likely**, the average time of QUICKSORT is $\Theta(n \lg n)$.

- Is is realistic to assume that all input permutations are equally likely?

  Not really. In many cases the input is almost sorted (e.g. rebuilding index in a database etc).

- If the assumption of uniform distribution is not realistic, then the fact that Quicksort has a good average time does not help in practice. How can we make QUICKSORT have a good average time irrespective of the input distribution?

  Using **randomization**!! See below.

5 Randomized algorithms

- So far the algorithms that we talked about are all *deterministic*. They do not make any random choices, and every time we run a deterministic algorithm we get the same output and same running time.

- A *randomized algorithm*, is an algorithm that make some random choices during its execution.

- Running time of a normal *deterministic* algorithm only depends on the input.

- Running time of a randomized algorithm depends not only on input but also on the random choices made by the algorithm.

- Running time of a randomized algorithm is not fixed for a given input!

- Randomized algorithms have best-case and worst-case running times, but the inputs for which these are achieved are not known, they can be any of the inputs.

- For a randomized algorithm we want to analyze its *expected* running time $T_e(n)$, that is, the expected (average) running time for all inputs of size n. Here $T(X)$ denotes the running time on input X (of size n), $T_e(n) = E_{|X|=n}[T(X)]$

6 Randomized Quicksort

- There are two ways to go about "randomizing" Quicksort: randomizing the permutation of the input, or randomizing the choice of the pivot.
- **Randomly permute the input:** The first idea is to randomly permute the input before running Quicksort, in order to enforce that all $n!$ permutations are equally likely. How?
  - Most computers have pseudo-random number generator $random(1, n)$ returning "random" integer between 1 and n
  - Algorithm: choose randomly an index k in $\{0, 1....., n-1\}$ and swap $A[0]$ with $A[k]$; then choose randomly an index k in $\{1....., n-1\}$ and swap $A[1]$ with $A[k]$; then choose randomly an index k in $\{2....., n-1\}$ and swap $A[2]$ with $A[k]$; and so on.

    This runs in $\Theta(n)$ time and generates an input permutation that's random (ie any of the $n!$ possible permutations of the input is equally likely).

- **Or, randomly choose the pivot:** Alternatively, we can modify PARTITION so that instead of picking the last element as pivot, it picks a random element as pivot. Here is a possible implementation':

```
RANDPARTITION(A, p, r)
i=RANDOM(p, r)
Exchange A[r] and A[i]
RETURN PARTITION(A, p, r)
```

```
RANDQUICKSORT(A, p, r)
IF p < r THEN
    q=RANDPARTITION(A, p, r)
    RANDQUICKSORT(A, p, q − 1)
    RANDQUICKSORT(A, q + 1, r)
```

- For both versions of Randomized Quicksort, the average (correctly: expected) running time is $\Theta(n \lg n)$ **no matter what the input distribution is**. That's pretty neat!

7 Final notes on Quicksort

- Quicksort is the fastest general-purpose sort. The constants involved in the $\Theta()$ term are small. Furthermore it iterates over the array in order which is efficient at the cache level. Being in-place avoids creating new arrays and gives it an advantage over Mergesort.

- Trivia: Quicksort was invented by Sir Charles Antony Richard "Tony" Hoare, a British computer scientist and winner of the 1980 Turing Award, in 1960, while he was a visiting student at Moscow State University. While in Moscow he was in the school of Kolmogorov (well-known mathematician).
- The original Quickort was described with a different partition algorithm, one that works from both ends; this is referd to as Hoare-partition.
- Both Hoare's and Lomuto partitions are in-place, but none of them is stable.
- Thus, in an efficient implementation, with an in-place partition, Quicksort is NOT stable.
- I don't know of any practical comparison of the two partitions, and I would guess they are both equally fast in practice. Lomuto-partition may be considered simpler and better, because it uses a single pointer (i, to point to the last element left of pivot)—and this is why textbooks prefer it. OTOH Hoare-partition does well (ie $O(n \lg n)$) when all elements are equal, while Lomuto-partition is quadratic.
- The idea behind Lomuto's partition can be extended to deal with elements equal to the pivot (place them in the middle instead of carrying them through recursion); also with 3-way partitioning.
- Quicksort can be made stable by using extra space (thus not in place anymore).
- In a future lecture (selection) we will see how to make quicksort run in worst-case $O(n \log n)$ time.

Appendix

Question: How does the array look to trigger worst-case for Quicksort?
Answer: sorted or reversely sorted