CS624 - Analysis of Algorithms

Quicksort

October 1, 2025

Quicksort

Algorithm 1 Quicksort(A,p,r)

if $p < r$ **then**
  $q \leftarrow Partition(A, p, r)$
  $Quicksort(A, p, q - 1)$
  $Quicksort(A, q + 1, r)$
end if

After the partition has been called the following is true:

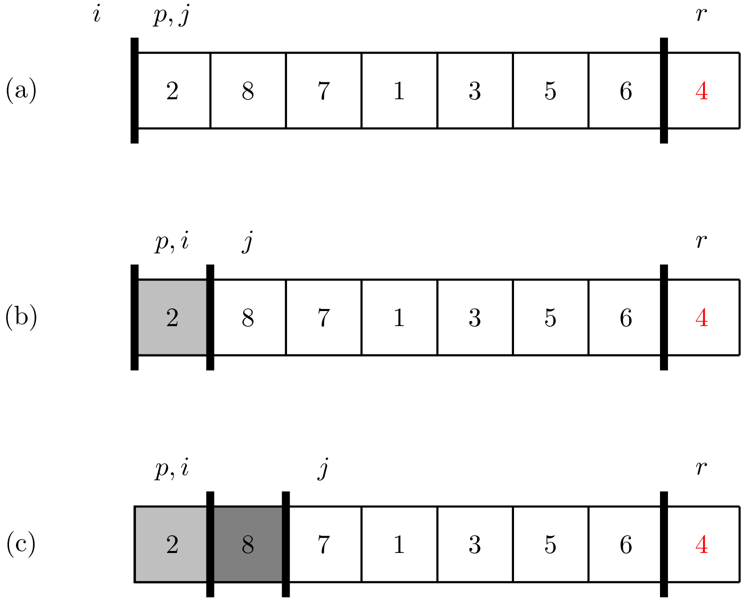
1. $p \leq q \leq r$.
2. The number $A[q]$ is in its final position. It will never be moved again.
3. If $i < q$, then $A[i] < A[q]$, and if $i > q$, then $A[i] > A[q]$.

Remember that q is the position of the pivot after partitioning.

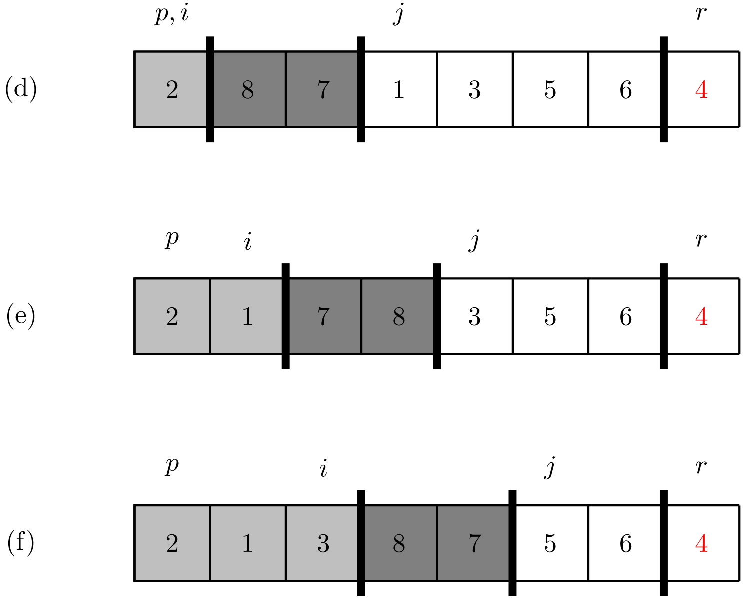
Algorithm 2 Partition(A,p,r)

```
x ← A[r] ▷x is the "pivot".
i ← p − 1 ▷i maintains the "left-right boundary".
for j ← p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1
        exchange A[i] ↔ A[j]
    end if
end for
exchange A[i + 1] ↔ A[r]
return  i + 1
```

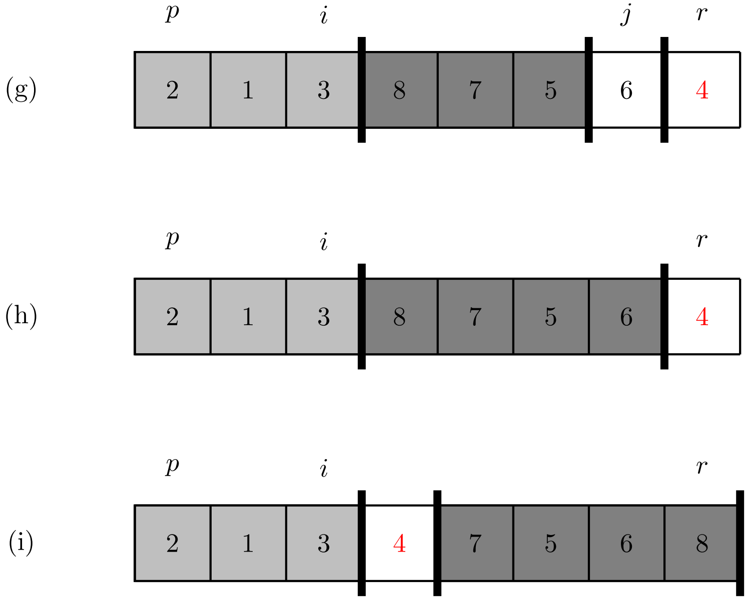
The Partition Method



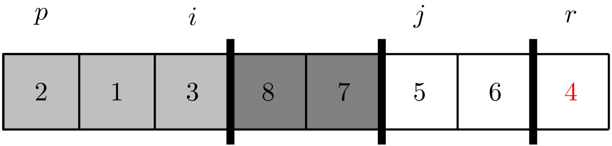
The Partition Method



The Partition Method



Partition, Proof of Correctness



Lemma

At the beginning of each iteration:

- $A[p..i]$ *are known to be* $\leq$ *pivot.*
- $A[i+1..j-1]$ *are known to be* $>$ *pivot.*
- $A[j, r-1]$ *not yet examined.*
- $A[r]$ *is the pivot.*

Proof.

Base: When we start out, j = p, i is p - 1, and the above are trivially true.

At the top of iteration j_0 of the for loop, i has the value i_0. Then by inductive hypothesis , at the top of that iteration of the for loop,

- All entries in $A[p..i_0]$ are $\leq$ *pivot*.
- All entries in $A[i_0 + 1..j_0 - 1]$ are $>$ *pivot*.
- $A[j_0..r - 1]$ consists of elements whose contents have not yet been examined.
- $A[r] = pivot$

□

Partition, Proof of Correctness, Case of $A[j_0] \leq pivot$

Proof.

- $A[j_0]$ and $A[i_0 + 1]$ are interchanged.
- $i_0 \to i_1 = i_0 + 1$, which is the value of i at the top of the next iteration of the for loop.

At the next iteration of the for loop $j \to j_1 = j_0 + 1$. Thus, since we interchanged $A[j_0]$ and $A[i_0 + 1]$, we have

- All entries in $A[p..i_1]$ are $\leq pivot$.
- All entries in $A[i_1 + 1..j_1 - 1]$ are $> pivot$. (These are the same elements that were originally in $A[i_0 + 1..j_0 - 1]$. The first one has been moved up to the end.)
- $A[j_1..r - 1]$ have not yet been examined.
- $A[r] = pivot$.

And this is just the inductive hypothesis at the top of the $j_0 + 1 = j_1$ iteration of the for loop. □

Partition, Proof of Correctness, Case of $A[j_0] > pivot$

Proof.

Nothing is done. At the next iteration of the for loop, we have

- $i_1 = i_0$ (because we didn't increment i).
- $j_1 = j_0 + 1$ (because we always increment j when we go to the next iteration).
- No change was made to the elements of the array A.

Thus, we have

- All entries in $A[p..i_1]$ continue to be $\leq pivot$.
- All entries in $A[i_1 + 1..j_1 - 1]$ are $>$the pivot. These are the original elements $A[i_0 + 1..j_0 - 1]$ plus $A[j_1 - 1] = A[j0]$
- $A[j_1..r - 1]$ have not yet been examined.
- $A[r] = pivot$

And this is just the the inductive hypothesis at the top of the $j_0 + 1 = j_1$ iteration of the for loop. This completes the proof. □

Partition, Proof of Correctness, Bottom Line

Proof.

- At the conclusion of the for loop, element r (which is the pivot element) is exchanged with element i+1 (which is the left-most element that is greater than the pivot element).
- This ensures that all the elements to the left of the pivot element have values $\leq$ the pivot, and all the elements to the right of the pivot element have values $>$ the pivot.

$\square$

Running Time – Best Case

- The runtime of partition is clearly $\Theta(n)$.
- The best case is when the array is partitioned into two equal parts.
- In this case the recurrence is $T(n) = 2T(n/2) + \Theta(n)$.
- We already know this is $\Theta(n \log n)$.

Running Time – Worst Case

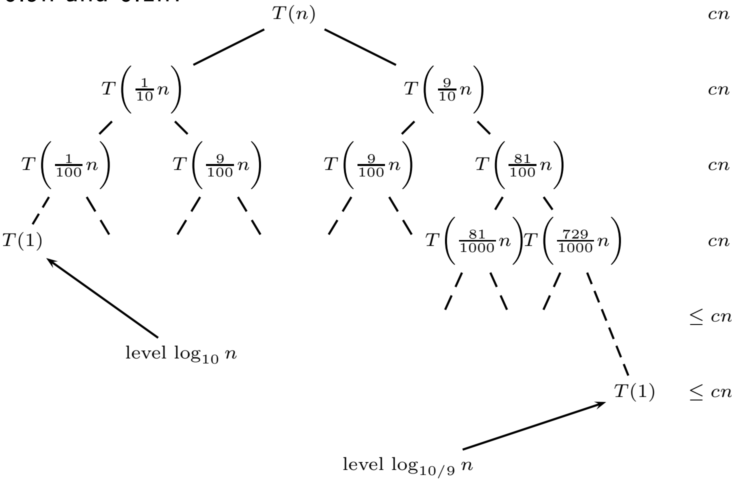
- The worst case happens when the pivot partitions the array into two sub arrays of size n-1 and 0.
- With our setting, this happens when the array is already sorted.
- Thus we have:

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + \Theta(n) = \sum_{j=0}^{n} \Theta(j) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)
\end{aligned}
$$

Running Time – Average Case

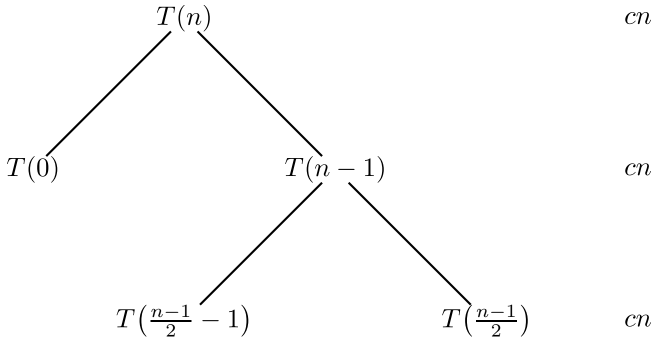
- We know the average runtime is $O(n \log n)$
- This means that on average we hit a "good" case.
- This is quite untypical, as usually the average case is no better than the worst case.

What happens if the pivot divides the array into two sub-arrays of 0.9n and 0.1n?



Running Time – Average Case

- There are $1 + \log_{(10/9)} n$ levels and each has $O(n)$ cost.
- The total cost is therefore $O(n \log n)$.
- In other words – quicksort is not THAT sensitive to the choice of pivot.
- But – the pivot is not always at the same relative position.
- What happens if occasionally it is as bad as can be?
- Suppose every other iteration the pivot is the largest element.



We simply double the number of levels, it is still $O(n\log(n))$

Randomized Analysis

- Remember the average runtime analysis of insertion sort.
- We averaged the running time over all possible inputs assuming they are all equally likely – random input, distributed uniformly.
- To do an average runtime analysis we have to know the distribution of the input.

Randomized Analysis

- Probabilistic analysis is the use of probability to analyze the runtime of an algorithm.
- It is used to calculate the average running time, assuming knowledge of the distribution of the input.
- A randomized algorithm is an algorithm that involves some randomness as part of its run.
- This doesn't mean the input is random.

Randomized Analysis

- We have a random number generator Random(p,r) which produces numbers between p and r, each with equal probability.
- The selected number is the pivot index.
- In practice most random algorithms produce pseudo-random numbers.
- When analyzing the running time of a randomized algorithm we take the expected run time over all inputs.

Randomized Quicksort

Define a function as follows:

Algorithm 3 RandomizedPartition(A,p,r)

1: $i \leftarrow Random(p, r)$
2: $Exchange A[i] \leftrightarrow A[r]$
3: **return** $Partition(A, p, r)$

Randomized Quicksort

Accordingly:

Algorithm 4 RandomizedQuicksort(A,p,r)

if $p < r$ **then**
 $q \leftarrow RandomizedPartition(A, p, r)$
 $RandomizedQuicksort(A, p, q - 1)$
 $RandomizedQuicksort(A, q + 1, r)$
end if

Rigorous Worst Case Analysis of Quicksort

- Let $T(n)$ be the worst case running time for quicksort (or randomized quicksort).
- We know there is a constant $a > 0$ such that $T(n) \leq \max_{0 \leq q \leq n-1} \big(T(q) + T(n-q-1) \big) + an$
- We know that probably $T(n) = O(n^2)$.
- This means there is a constant c such that $T(n) \leq cn^2$.

Rigorous Worst Case Analysis of Quicksort

Proof by induction.

- This is certainly true for k=1.
- Suppose this is true for all $k < n$ with some fixed constant c.

$$T(n) \le \max_{0 \le q \le n-1} \big(T(q) + T(n-q-1) \big) + an$$
$$\le c \max_{0 \le q \le n-1} \left(q^2 + (n-q-1)^2 \right) + an$$

- The expression $\left(q^2 + (n-q-1)^2\right)$ is a convex function, achieving a maximum at the endpoints – 0 and n-1.
- In those endpoints the value is $(n-1)^2$.

$\square$

Rigorous Worst Case Analysis of Quicksort

Proof by induction, Cont.

- Therefore:

$$\begin{aligned}
T(n) &\leq \max_{0\leq q\leq n-1}\big(T(q) + T(n-q-1)\big) + an \\
&\leq c \max_{0\leq q\leq n-1}\big(q^2 + (n-q-1)^2\big) + an \\
&\leq cn^2 - c(2n-1) + an \\
&= cn^2 - (2c-a)n + c \\
&\leq cn^2 - (2c-a)n + cn \quad \dagger \\
&= cn^2 - (c-a)n
\end{aligned}$$

- Assuming $n \geq 1$ and picking a large enough c so that $c \geq a$.

□

† Here's where we use the assumption that $n \geq 1$.

Rigorous Worst Case Analysis of Quicksort

- The above gives an upper bound to the worst case runtime.
- Previously we have seen a case where the runtime is quadratic.
- That's when the pivot always divides the array into n-1 and 0 sub-arrays.
- We now saw that $T(n) = O(n^2)$.
- So in the worst case $T(n) = \Theta(n^2)$.

Average Case Analysis – Method 1

- It is easy to use Randomized-Quicksort.
- Let T(n) be the average runtime for an array of size n:

$$T(n) = \frac{1}{n} \sum_{q=0}^{n-1} \big(T(q) + T(n-q-1)\big) + cn + \Theta(1).$$

- Which is actually $T(n) = \frac{2}{n} \sum_{q=0}^{n-1} T(q) + cn + \Theta(1).$
- We wrote $cn + \Theta(1)$ rather than $\Theta(n)$ since we can assume we do "everything" every time we call Partition.
- This is a worst case assumption that allows us to do something really nice mathematically.

Average Case Analysis – Method 1

- Multiplying by n we get: $nT(n) = 2\sum_{q=0}^{n-1} T(q) + cn^2 + \Theta(n)$
- Multiplying by $n+1$ we get:
  $(n+1)T(n+1) = 2\sum_{q=0}^{n} T(q) + c(n+1)^2 + \Theta(n)$
- Subtracting the two cancels most terms out:
  $(n+1)T(n+1) - nT(n) = 2T(n) + \Theta(n)$

Average Case Analysis – Method 1

- Collecting terms: $(n+1)T(n+1) = (n+2)T(n) + \Theta(n)$
- Dividing by $(n+1)(n+2)$ we get: $\frac{T(n+1)}{n+2} = \frac{T(n)}{n+1} + \Theta\left(\frac{1}{n}\right)$
- Defining $g(n) = \frac{T(n)}{(n+1)}$: $g(n+1) = g(n) + \Theta\left(\frac{1}{n}\right)$
- Thus: $g(n) = \Theta\left(\sum_{k=1}^{n-1} \frac{1}{k}\right) = \Theta(\log n)$
- Going back to T: $T(n) = (n+1)g(n) = \Theta(n \log n)$

Average Case Analysis – Method 2

- The total cost = the sum of the costs of all the calls to RandomizedPartition.
- The cost of a call to RandomizedPartition is O(No. for loop executions) which is O(No. comparisons).
- The expected cost of RandomizedQuicksort is O(expected number of comparisons).
- Notice that once a key x_k is chosen as pivot, the elements to its left will never be compared to the elements to its right.

Average Case Analysis – Method 2

- Consider $\{x_i, x_{i+1}, ..., x_{j-1}, x_j\}$, the set of keys in sorted order.
- Any two keys here are compared only if one of them is pivot and that is the last time they are all in the same partition.
- Each key is equally likely to be chosen.
- x_i and x_j can be compared only if one of them is pivot and this will only happen if this is the first pivot from the set $\{x_i, x_{i+1}, ..., x_{j-1}, x_j\}$.
- The probability of this is $\frac{2}{(j-i+1)}$.

Average Case Analysis – Method 2

The expected number of comparisons is:

$$\sum_{i<j} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$\leq \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = 2(n-1)H_n = O(n \log n)$$

The Hiring Problem

- You need an office assistant and want to minimize the hiring cost
- An agency sends one candidate every day – at a fixed cost C_i
- Interview the person, either hire him/her at a cost of $C_h >> C_i$ (and fire the old one), or keep old one.
- We always want the best person – hire if interviewee is better than current person.
- Assume we have a metric that can always determine, given two candidates, who is better, and that no two candidates are equally good.
- The candidates are sent in a random order.

The Hiring Problem

Define a function as follows:

Algorithm 5 Hire-Candidate(n)

bestCandidate $\leftarrow 0$ // Dummy candidate
for $j \leftarrow 1..n$ **do**
    Pay C_i
    Interview Candidate j
    if i better than best-candidate **then**
        Pay C_h
        Hire Candidate j
        bestCandidate $\leftarrow j$
    end if
end for

The Hiring Problem – Analysis

- The total cost is $n * C_i + m * C_h$ where m is the number of candidates we hire.
- Assume applicants come in random order – each permutation of applicants is equally likely.
- What is the worst case?

The Hiring Problem – Analysis

- The total cost is $n * C_i + m * C_h$ where m is the number of candidates we hire.
- Assume applicants come in random order – each permutation of applicants is equally likely.
- What is the worst case?
- We hire every candidate! So they are sorted in increasing order of quality.

The Hiring Problem – Analysis

- The total cost is $n * C_i + m * C_h$ where m is the number of candidates we hire.
- Assume applicants come in random order – each permutation of applicants is equally likely.
- What is the worst case?
- We hire every candidate! So they are sorted in increasing order of quality.
- What is the best case?

The Hiring Problem – Average Case

- Let us define a random variable X_i as follows:

$$X_i = \begin{cases} 1 & X_i \text{ is hired} \\ 0 & X_i \text{ is not hired} \end{cases}$$

- This is called an *indicator random variable* and it is good at analyzing sequences of random trials.
- Let X be the random variable whose value equals the number of times we hire a new assistant.
- The expected value of X is $E[X] = \sum_{i=1}^{n} Pr\{X = x\}$
- Using indicator random variables, $E[X_i] = Pr\{\text{Candidate i is hired}\}$

The Hiring Problem – Average Case

- Candidate i is hired when s/he is better than all previous i-1 candidates.
- We assume candidates arrive in random order.
- Any one of the first i candidates is equally likely to be the most qualified so far.
- Therefore, candidate i has a 1/i probability of being the most qualified so far, and hence a 1/i probability of being hired.
- So, $E[X_i] = \frac{1}{i}$.
- $E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{i} = ln(n) + O(1)$
- It is the harmonic series.
- The expected hiring cost is therefore $O(C_h \ln n)$.