Intuition for the average case

• The behavior of quicksort depends on the relative ordering of the values in the input array, not by the particular values in the array.

• We assume for now that all permutations of the input numbers are equally likely.

• On a random input, partitioning is highly unlikely to happen in the same way at every level, as our informal analysis has assumed.

• Some splits will be reasonably well balanced and some will be fairly unbalanced.

• In the average case, \textsc{Partition} produces a mix of “good” and “bad” splits.

• In an average-case, the good and bad splits are distributed randomly throughout the tree.

• Suppose, that the best-case and worst-case splits alternate in the tree.

• At the root, the cost is $n$ for partitioning, and the subarrays produced have sizes $n-1$ and $0$.

• At the next level, the subarray of size $n-1$ undergoes best-case partitioning.

• Let’s assume that the boundary-condition cost is 1 for the subarray of size 0.
The combination of the bad split followed by the good split produces three subarrays of sizes 0, \((n - 1)/2 - 1\), and \((n - 1)/2\) at a combined cost of \(\Theta(n) + \Theta(n - 1) = \Theta(n)\). Certainly, this situation is no worse than a single level of partitioning that produces two subarrays of size \((n - 1)/2\) at a cost of \(\Theta(n)\). Yet the latter situation is balanced!

Intuitively, the \(\Theta(n - 1)\) cost of the bad split can be absorbed into the \(\Theta(n)\) cost of the good split, and the resulting split is good. Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone. Still \(O(n \log n)\), but with a slightly larger constant hidden by the \(O\)-notation.
7.3 A randomized version of quicksort

- For quicksort random sampling, yields a simpler analysis than permuting the input
- We select a randomly chosen element from the $A[p..r]$ as the pivot
- We first exchange element $A[r]$ with an element chosen at random
- By randomly sampling the range $p,...,r$, we ensure that the pivot element is equally likely to be any of the $r - p + 1$ elements

Because of the random pivot element, we expect the split of the input array to be reasonably well balanced on average

In the new PARTITION procedure, we simply implement the swap before actually partitioning

```
RANDOMIZED-PARTITION($A, p, r$)
1. $i = \text{RANDOM}(p, r)$
2. exchange $A[r]$ with $A[i]$
3. return PARTITION($A, p, r$)
```
7.4 Analysis of quicksort

- A worst-case split at every level of recursion produces a $\Theta(n^2)$ running time, which, intuitively, is the worst-case running time.
- Using the substitution method we can show that the running time of quicksort is $O(n^2)$.
- Let $T(n)$ be the worst-case time for QUICKSORT on an input of size $n$:
  $$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n)$$
Parameter $q$ ranges from 0 to $n - 1$ because PARTITION produces two subproblems with total size $n - 1$

We guess that $T(n) \leq cn^2$ for some $c$

Substituting the guess gives

$T(n) \leq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n)$

$= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n)$

$q^2 + (n - q - 1)^2$ achieves a maximum over $0 \leq q \leq n - 1$ at either endpoint

- The second derivative of the expression with respect to $q$ is positive

This observation gives us the bound

$\max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) \leq (n - 1)^2$

$= n^2 - 2n + 1$

Continuing with bounding of $T(n)$:

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

We can pick the constant $c$ large enough so that term $c(2n - 1)$ dominates term $\Theta(n)$

Thus, $T(n) = O(n^2)$

Recurrence above has solution $T(n) = \Omega(n^2)$

Thus, the (worst-case) running time of quicksort is $\Theta(n^2)$
7.4.2 Expected running time

- The intuition why the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$
  - if the split induced by the algorithm puts any constant fraction of the elements on one side of the partition, then the recursion tree
    - has depth $\Theta(\lg n)$, and
    - $O(n)$ work is performed at each level
- Even if we add a few new levels with the most unbalanced split possible, the total time remains $O(n \lg n)$

- Each time PARTITION is called, it selects a pivot element which is never included in any future recursive calls
- There can be at most $n$ calls to PARTITION over execution of the quicksort algorithm
- Call to PARTITION takes $O(1) +$ time proportional to the number of iterations of the for loop in lines 3–6
- Each iteration of this for loop performs a comparison in line 4, comparing the pivot element to another element of the array $A$
If we can count the total number of times that line 4 is executed, we can bound the total time spent in the for loop during the entire execution of QUICKSORT.

**Lemma 7.1**

*Let $X$ be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an $n$-element array. Then the running time of QUICKSORT is $O(n + X)$.*

Our goal is to compute $X$.

- We do not analyze how many comparisons are made in a call to PARTITION, but derive an bound on the total number of comparisons.
- When the algorithm compares two elements of the array and when it does not?
- For ease of analysis, we rename the elements of the array $A$ as $z_1, z_2, \ldots, z_n$, with $z_i$ being the $i$th smallest element.
- Let the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ be the set of elements between $z_i$ and $z_j$, inclusive.
• When does the algorithm compare $z_i$ and $z_j$?
• First observe that each pair of elements is compared at most once.
• Elements are compared only to the pivot and, after a particular call to PARTITION, the pivot element used in that call is never again compared to any other elements.
• We define $X_{ij} = I\{z_i \text{ is compared to } z_j\}$, where we are considering whether the comparison takes place at any time during the execution of the algorithm.

Each pair is compared at most once, easily characterize the number of comparisons:

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

• Taking expectations, using linearity of expectation, and Lemma 5.1, we obtain

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_i\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_i]$$

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$
We need to compute $\Pr\{z_i \text{ is compared to } z_j\}$

We assume that RANDOMIZED-PARTITION chooses pivots randomly and independently.

Consider an input of the numbers 1 through 10, and suppose that the first pivot is 7.

The call to PARTITION separates the numbers into two sets: \{1,2,3,4,5,6\} and \{8,9,10\}.

In doing so, the pivot element 7 is compared to all other elements, but no number from the first set (e.g., 2) is or ever will be compared to any number from the second set (e.g., 9).

We assume that element values are distinct.

Once a pivot $x$ is chosen with $z_i < x < z_j$, we know that $z_i$ and $z_j$ cannot be compared at any subsequent time.

If $z_i$ is chosen as a pivot before any other item in $Z_{ij}$, then $z_i$ will be compared to each item in $Z_{ij}$, except for itself.

Similarly, if $z_j$ is chosen as a pivot before any other item in $Z_{ij}$, then $z_j$ will be compared to each item in $Z_{ij}$, except for itself.
In our example, the values 7 and 9 are compared because 7 is the first item from $Z_{79}$ to be chosen as a pivot.

In contrast, 2 and 9 will never be compared because the first pivot element chosen from $Z_{29}$ is 7.

Thus, $z_i$ and $z_j$ are compared if and only if the first element to be chosen as a pivot from $Z_{ij}$ is either $z_i$ or $z_j$.

Prior to the point at which an element from $Z_{ij}$ has been chosen as a pivot, the whole set $Z_{ij}$ is together in the same partition.

Therefore, any element of $Z_{ij}$ is equally likely to be the first one chosen as a pivot.

Because the set $Z_{ij}$ has $j - i + 1$ elements, and because pivots are chosen randomly and independently, the probability that any given element is the first one chosen as a pivot is $1/(j - i + 1)$.
\[ \Pr\{Z_i \text{ is compared to } z_j \} = \Pr\{Z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} = \Pr\{Z_i \text{ is first pivot chosen from } Z_{ij}\} + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} = \frac{1}{j - i + 1} + \frac{1}{j - i + 1} = \frac{2}{j - i + 1} \]

- The third line follows because the two events are mutually exclusive.

\[ \Pr\{Z_i \text{ is compared to } z_j \} = \Pr\{Z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} = \Pr\{Z_i \text{ is first pivot chosen from } Z_{ij}\} + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} = \frac{1}{j - i + 1} + \frac{1}{j - i + 1} = \frac{2}{j - i + 1} \]

- Combining, we get that

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

- We can evaluate this sum using a change of variables \( k = j - i \) and the bound on the harmonic series

\[ E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n) \]
8 Sorting in Linear Time

- The sorting algorithms introduced thus far are **comparison sorts**
- Any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort $n$ elements
- Merge sort and heapsort are asymptotically optimal
- We examine counting sort, radix sort, and bucket sort that run in linear time

8.1 Lower bounds for sorting

- We can view comparison sorts abstractly in terms of **decision trees**
  - a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm
• The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf
• An internal node compares \( a_i \leq a_j \)
• When we come to a leaf, the sorting algorithm has established the ordering
  \[ a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)} \]
• Any correct sorting algorithm must be able to produce each permutation of its input
  – Each of the \( n! \) permutations must appear as one of the leaves of the decision tree for a comparison sort to be correct

A lower bound for the worst case

• Length of the longest path from the root to any of the reachable leaves is the worst-case number of comparisons
• The worst-case number of comparisons for a given algorithm equals the height of its tree
• A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is a bound on the running time of any comparison sort algorithm
**Theorem 8.1**

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

**Proof** It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.

Consider a decision tree of height $h$ with $l$ reachable leaves corresponding to a comparison sort on $n$ elements.

Each of the $n!$ permutations of the input appears as some leaf.

Therefore, we have $n! \leq l$.

Since a binary tree of height $h$ has no more than $2^h$ leaves, we have

$$n! \leq l \leq 2^h$$

which, by taking logarithms, implies $h \geq \lg(n!)$ since the $\lg$ function is monotonically increasing.

Furthermore, $h = \Omega(n \lg n)$, because $\lg(n!) = \Theta(n \lg n)$

• Hence, heapsort and merge sort are asymptotically optimal comparison sorts
8.2 Counting sort

- Assume that each input element is an integer in the range \(0\) to \(k\), for some integer \(k\)
- When \(k = O(n)\), the sort runs in \(\Theta(n)\) time
- Counting sort determines, for an input element \(x\), the number of elements < \(x\)
- It uses this information to place element \(x\) directly into its position in the output array
- For example, if 17 elements are less than \(x\), then \(x\) belongs in output position 18

In addition to the input array \(A[1..n]\) we require two other arrays:
- \(B[1..n]\) holds the sorted output, and
- \(C[0..k]\) provides temporary working storage
How much time does counting sort require?

- The for loop of lines 2–3 takes time $\Theta(k)$
- the for loop of lines 4–5 takes time $\Theta(n)$
- the for loop of lines 7–8 takes time $\Theta(k)$, and
- the for loop of lines 10–12 takes time $\Theta(n)$

Thus, the overall time is $\Theta(k + n)$

In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$
Counting sort beats the lower bound of $\Omega(n \lg n)$ because it is not a comparison sort.

In fact, no comparisons between input elements occur anywhere in the code.

Instead, counting sort uses the actual values of the elements to index into an array.

An important property of counting sort is that it is **stable**.

– numbers with the same value appear in the output array in the same order as they do in the input array.

---

8.3 Radix sort

Radix sort is used by the card-sorting machines you now find only in computer museums.

Radix sort solves the problem of card sorting counter-intuitively by sorting on the least significant digit first.

In order for radix sort to work correctly, the digit sorts must be stable.
Radix-Sort \((A, d)\)

1. for \(i = 1\) to \(d\)
2. use a stable sort to sort array \(A\) on digit \(i\)

Lemma 8.3  Given \(n\) \(d\)-digit numbers in which each digit can take on up to \(k\) possible values, Radix-Sort sorts the numbers in \(\Theta(d(k + n))\) time if the stable sort takes \(\Theta(k + n)\) time.

Proof  The correctness of radix sort follows by induction on the columns. When each digit is in the range \(0\) to \(k - 1\) (so that it can take on \(k\) possible values), and \(k\) is not too large, counting sort is the obvious choice. Each pass over \(n\) \(d\)-digit numbers then takes time \(\Theta(k + n)\). There are \(d\) passes, and so the total time for radix sort follows.
8.4 Bucket sort

- Assume that the input is drawn from uniform distribution and has an average-case running time of $O(n)$
- Counting and bucket sort are fast because they assume something about the input
  - Counting sort: the input contains integers in a small range,
  - Bucket sort: the input is drawn from a random process that distributes elements uniformly and independently over the interval $[0, 1)$

Bucket sort divides the interval $[0, 1)$ into $n$ equal-sized subintervals, or buckets, and then distributes the $n$ input numbers into the buckets

- Since inputs are uniformly and independently distributed over $[0, 1)$, we do not expect many numbers to fall into each bucket
- We sort numbers in each bucket and go through the buckets in order, listing the elements in each
- The input is an $n$-element array $A$ and each element $A[i]$ in the array satisfies $0 \leq A[i] < 1$
- The code requires an auxiliary array $B[0..n-1]$ of linked lists (buckets)
**BUCKET-SORT**($A$)

1. let $B[0..n-1]$ be a new array
2. $n = A.length$
3. **for** $i = 0$ to $n-1$
4. make $B[i]$ an empty list
5. **for** $i = 1$ to $n$
7. **for** $i = 0$ to $n-1$
8. sort list $B[i]$ with insertion sort
9. concatenate the lists $B[0], B[1], ..., B[n-1]$ together in order
Consider two elements $A[i]$ and $A[j]$

Assume wlog that $A[i] \leq A[j]$

Since $[nA[i]] \leq [nA[j]]$, either element $A[i]$ goes into the same bucket as $A[j]$ or it goes into a bucket with a lower index.

If $A[i]$ and $A[j]$ go into the same bucket
  - the for loop of lines 7–8 puts them into the proper order.

If $A[i]$ and $A[j]$ go into different buckets
  - line 9 puts them into the proper order.

Therefore, bucket sort works correctly.

Observe that all lines except line 8 take $O(n)$ time in the worst case.

We need to analyze the total time taken by the $n$ calls to insertion sort in line 8.

Let $n_i$ be the random variable denoting the number of elements placed in bucket $B[i]$.

Since insertion sort runs in quadratic time, the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$
We now compute the expected value of the running time, where we take the expectation over the input distribution.

Taking expectations of both sides and using linearity of expectation, we have

\[ E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \]

\[ = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \]

Claim: \( E[n_i^2] = 2 - 1/n \) for \( i = 0, 1, \ldots, n - 1 \)

It is no surprise that each bucket \( i \) has the same value of \( E[n_i^2] \), since each value in \( A \) is equally likely to fall in any bucket.

To prove the claim, define indicator random variables \( X_{ij} = I\{A[j] \text{ falls in bucket } i\} \) for \( i = 0, 1, \ldots, n - 1 \) and \( j = 1, 2, \ldots, n \)

Thus,

\[ n_i = \sum_{j=1}^{n} X_{ij} \]
\[
E[n_i^2] = E \left[ \left( \sum_{j=1}^{n} X_{ij} \right)^2 \right] = E \left[ \sum_{j=1}^{n} \sum_{k=1}^{n} X_{ij} X_{ik} \right] \\
= E \left[ \sum_{j=1}^{n} X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} X_{ij} X_{ik} \right] \\
= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{1 \leq k \leq n, k \neq j} E[X_{ij} X_{ik}] \\
\]

- Indicator random variable \( X_{ij} \) is 1 with probability \( \frac{1}{n} \) and 0 otherwise, therefore
  \[
  E \left[ X_{ij}^2 \right] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left( 1 - \frac{1}{n} \right) = \frac{1}{n} 
  \]
- When \( k \neq j \), the variables \( X_{ij} \) and \( X_{ik} \) are independent, and hence
  \[
  E[X_{ij} X_{ik}] = E[X_{ij}] E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2} 
  \]
Substituting these two expected values, we obtain

\[ E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n} \frac{1}{n^2} \]

\[ = n \cdot \frac{1}{n} + n(n - 1) \cdot \frac{1}{n^2} \]

\[ = 1 + \frac{n - 1}{n} = 2 - \frac{1}{n} \]

which proves the claim.

Using this expected value we conclude that average-case running time for bucket sort is

\[ \Theta(n) + n \cdot O\left(2 - 1/n\right) = \Theta(n) \]

Even if the input is not drawn uniformly, bucket sort may still run in linear time.

As long as the input has the property

– the sum of the squares of the bucket sizes is linear in the total number of elements,

bucket sort will run in linear time.