Quicksort

Revision of September 11, 2014
Reference: Chapter 7 of CLRS
Reference: Chapter 7 of CLRS

Outline:
- Partitions
- Quicksort
- Analysis of Quicksort
Given: An array of numbers
Partition: Rearrange the array \( A[p..r] \) in place into two (possibly empty) subarrays \( A[p..q-1] \) and \( A[q+1..r] \) such that

\[ A[u] < A[q] < A[v], \quad \text{for any } p \leq u \leq q - 1 \text{ and } q + 1 \leq v \leq r \]

\[ p \quad \underbrace{\quad \underbrace{x} \quad \overbrace{\quad x} \quad \overbrace{\cdots \quad r}} \quad \underbrace{\quad \underbrace{x} \quad \overbrace{\quad x} \quad \overbrace{\cdots \quad r}} \]

\[ x = A[r] \]
Given: An array of numbers
Partition: Rearrange the array $A[p..r]$ in place into two (possibly empty) subarrays $A[p..q - 1]$ and $A[q + 1..r]$ such that

$$A[u] < A[q] < A[v], \quad \text{for any } p \leq u \leq q - 1 \text{ and } q + 1 \leq v \leq r$$

$x = A[r]$

$x$ is called the pivot. Assume $x = A[r]$; if not, swap first
Given: An array of numbers
Partition: Rearrange the array $A[p..r]$ in place into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that

$$A[u] < A[q] < A[v], \quad \text{for any } p \leq u \leq q - 1 \text{ and } q + 1 \leq v \leq r$$

$x$ is called the pivot. Assume $x = A[r]$; if not, swap first

Quicksort works by:
Given: An array of numbers
Partition: Rearrange the array $A[p..r]$ in place into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that

$$A[u] < A[q] < A[v], \quad \text{for any } p \leq u \leq q-1 \text{ and } q+1 \leq v \leq r$$

$x$ is called the pivot. Assume $x = A[r]$; if not, swap first

Quicksort works by:

1. calling partition first
Given: An array of numbers
Partition: Rearrange the array $A[p..r]$ in place into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that

$$A[u] < A[q] < A[v], \quad \text{for any } p \leq u \leq q-1 \text{ and } q+1 \leq v \leq r$$

$x = A[r]$ is called the pivot. Assume $x = A[r]$; if not, swap first
Quicksort works by:

1. calling partition first
2. recursively sorting $A[\quad]$ and $A[\quad]$
**Partition**

**Given:** An array of numbers

**Partition:** Rearrange the array $A[p..r]$ in place into two (possibly empty) subarrays $A[p..q - 1]$ and $A[q + 1..r]$ such that

$$A[u] < A[q] < A[v], \text{ for any } p \leq u \leq q - 1 \text{ and } q + 1 \leq v \leq r$$

$x = A[r]$ is called the **pivot**. Assume $x = A[r]$; if not, swap first

**Quicksort** works by:

1. calling partition first
2. recursively sorting $A[p..q - 1]$ and $A[ ]$
Given: An array of numbers

Partition: Rearrange the array $A[p..r]$ in place into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that

$$A[u] < A[q] < A[v], \text{ for any } p \leq u \leq q - 1 \text{ and } q + 1 \leq v \leq r$$

$A[r]$ is called the pivot. Assume $x = A[r]$; if not, swap first

Quicksort works by:

1. calling partition first
2. recursively sorting $A[p..q-1]$ and $A[q+1..r]$
Partitioning $A[p..r]$ with extra memory

- Copy $A[p..r]$ to another array $B[p..r]$
Partitioning $A[p..r]$ with extra memory

- Copy $A[p..r]$ to another array $B[p..r]$

- With $p - r$ comparisons find the rank $R$ of $x = A[r]$ in $B[p..r]$
Partitioning $A[p..r]$ with extra memory

- Copy $A[p..r]$ to another array $B[p..r]$

- With $p - r$ comparisons find the rank $R$ of $x = A[r]$ in $B[p..r]$

- Copy the items in $B[p..r]$ back to $A[p..r]$ placing
  - items smaller than $x$ into first $R - 1$ locations
  - $x$ into location $p + R - 1$
  - items larger than $x$ into last $r - R$ locations
Partitioning $A[p..r]$ with extra memory

- Copy $A[p..r]$ to another array $B[p..r]$

- With $p - r$ comparisons find the rank $R$ of $x = A[r]$ in $B[p..r]$

- Copy the items in $B[p..r]$ back to $A[p..r]$ placing
  - items smaller than $x$ into first $R - 1$ locations
  - $x$ into location $p + R - 1$
  - items larger than $x$ into last $r - R$ locations

- $O(r - p)$ time but needs extra space.
Use $A[r]$ as the pivot, and grow partition from left to right. 

$i$ will be largest index of processed item $\leq x$. 

$j$ will be smallest index of unprocessed item.

Initially $(i, j) = (p - 1, p)$

Increase $j$ by 1 each time to find a place for $A[j]$. At the same time increase $i$ when necessary.

Stops when $j = r$
Use $A[r]$ as the pivot, and grow partition from left to right.

- $i$ will be largest index of processed item $\leq x$.
- $j$ will be smallest index of unprocessed item.

Initially $(i, j) = (p - 1, p)$
Partition($A, p, r$) without extra memory

Use $A[r]$ as the pivot, and grow partition from left to right.

- $i$ will be largest index of processed item $\leq x$.
- $j$ will be smallest index of unprocessed item.

1. Initially $(i, j) = (p - 1, p)$
2. Increase $j$ by 1 each time to find a place for $A[j]$.
   At the same time increase $i$ when necessary.
Use $A[r]$ as the pivot, and grow partition from left to right. 

- $i$ will be largest index of processed item $\leq x$.
- $j$ will be smallest index of unprocessed item.

1. Initially $(i, j) = (p - 1, p)$
2. Increase $j$ by 1 each time to find a place for $A[j]$
   At the same time increase $i$ when necessary
3. Stops when $j = r$
One Iteration of the Procedure Partition

Increase $j$ by 1 each time to find a place for $A[j]$

At the same time increase $i$ when necessary
One Iteration of the Procedure Partition

Increase $j$ by 1 each time to find a place for $A[j]$

At the same time increase $i$ when necessary

(A) $A[j] > x$

(B) $A[j] \leq x$
One Iteration of the Procedure Partition

Increase $j$ by 1 each time to find a place for $A[j]$

At the same time increase $i$ when necessary

(A) $A[j] > x$

(B) $A[j] \leq x$

1. Only increase $j$ by 1
One Iteration of the Procedure Partition

Increase $j$ by 1 each time to find a place for $A[j]$.

At the same time increase $i$ when necessary.

1. Only increase $j$ by 1
2. $i = i + 1$. 

(A) $A[j] > x$

(B) $A[j] \leq x$
Increase \( j \) by 1 each time to find a place for \( A[j] \)

At the same time increase \( i \) when necessary

1. Only increase \( j \) by 1
2. \( i = i + 1 \). \( A[i] \leftrightarrow A[j] \).
One Iteration of the Procedure Partition

Increase \( j \) by 1 each time to find a place for \( A[j] \)

At the same time increase \( i \) when necessary

1. Only increase \( j \) by 1
2. \( i = i + 1. \) \( A[i] \leftrightarrow A[j]. \) \( j = j + 1 \)
Example: The Operation of Partition($A, p, r$)

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p, j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>(1)</td>
</tr>
<tr>
<td>$p, i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>(2)</td>
</tr>
<tr>
<td>$p, i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>$p, i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>(4)</td>
</tr>
<tr>
<td>$p, i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>(5)</td>
</tr>
<tr>
<td>$p, i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>(6)</td>
</tr>
<tr>
<td>$p, i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>(7)</td>
</tr>
<tr>
<td>$p, i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>(8)</td>
</tr>
<tr>
<td>$p, i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>(9)</td>
</tr>
</tbody>
</table>
Partition($A, p, r$)

\begin{verbatim}
begin
    x = A[r]; // A[r] is the pivot element
end
\end{verbatim}
The Partition($A, p, r$) Algorithm

Partition($A, p, r$)

begin
    $x = A[r]$; // $A[r]$ is the pivot element
    $i = p - 1$;
    for $j = p$ to $r - 1$ do
        if $A[j] \leq x$ then
            $i = i + 1$;
            exchange $A[i]$ and $A[j]$;
        end
    exchange $A[i + 1]$ and $A[r]$; // put pivot in position
end

return $i + 1$ // $q = i + 1$
The Partition($A, p, r$) Algorithm

Partition($A, p, r$)

begin
    $x = A[r];$ // $A[r]$ is the pivot element
    $i = p - 1;$
    for $j = p$ to $r - 1$ do
        if $A[j] \leq x$ then
            $i = i + 1;$
            exchange $A[i]$ and $A[j]$;
        end
    end

    exchange $A[i + 1]$ and $A[r]$;
    // put pivot in position

    return $i + 1$ // $q = i + 1$
The Partition($A, p, r$) Algorithm

**Partition($A, p, r$)**

```plaintext
begin
  \( x = A[r]; \) // \( A[r] \) is the pivot element
  \( i = p - 1; \)
  for \( j = p \) to \( r - 1 \) do
    if \( A[j] \leq x \) then
      \( i = i + 1; \)
      exchange \( A[i] \) and \( A[j] \);
  end
end
```
The Partition($A, p, r$) Algorithm

Partition($A, p, r$)

begin
  $x = A[r]$; // $A[r]$ is the pivot element
  $i = p - 1$;
  for $j = p$ to $r - 1$ do
    if $A[j] \leq x$ then
      $i = i + 1$;
      exchange $A[i]$ and $A[j]$;
    end
  end
end

; // put pivot in position
The Partition\((A, p, r)\) Algorithm

\textbf{Partition}(A, p, r)

\begin{algorithm}
begin
\hspace{1em} x = A[r]; // A[r] is the pivot element
\hspace{1em} i = p - 1;
\hspace{1em} for j = p to r - 1 do
\hspace{2em} if A[j] \leq x then
\hspace{3em} i = i + 1;
\hspace{3em} exchange A[i] and A[j];
\hspace{2em} end
\hspace{1em} end
\hspace{1em} exchange A[i + 1] and A[r]; // put pivot in position
end
\end{algorithm}
The Partition($A, p, r$) Algorithm

Partition($A, p, r$)

begin

   $x = A[r]$; // $A[r]$ is the pivot element
   $i = p - 1$;

   for $j = p$ to $r - 1$ do
      if $A[j] \leq x$ then
         $i = i + 1$;
         exchange $A[i]$ and $A[j]$;
      end
   end

   exchange $A[i + 1]$ and $A[r]$; // put pivot in position

return
The Partition($A, p, r$) Algorithm

Partition($A, p, r$)

begin
    $x = A[r]$; // $A[r]$ is the pivot element
    $i = p - 1$;
    for $j = p$ to $r - 1$ do
        if $A[j] \leq x$ then
            $i = i + 1$;
            exchange $A[i]$ and $A[j]$;
        end
    end
    exchange $A[i + 1]$ and $A[r]$; // put pivot in position
    return $i + 1$ // $q = i + 1$
end
Running Time of Partition\((A, p, r)\)

**Partition\((A, p, r)\)**

\begin{verbatim}
begin
    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]; // O(r - p)
        end
    end
    exchange A[i + 1] and A[r];
return i + 1
end
\end{verbatim}

Running time is \(O(r - p)\) linear in the length of the array \(A[p..r]\)
Partition \((A, p, r)\)

\[
\begin{array}{l}
\text{begin} \\
\quad x = A[r]; \\
\quad i = p - 1; \\
\quad \textbf{for } j = p \textbf{ to } r - 1 \textbf{ do} \\
\quad \quad \textbf{if } A[j] \leq x \textbf{ then} \\
\quad \quad \quad i = i + 1; \\
\quad \quad \quad \text{exchange } A[i] \text{ and } A[j]; \quad // \quad O(r - p) \\
\quad \textbf{end} \\
\textbf{end} \\
\text{exchange } A[i + 1] \text{ and } A[r]; \\
\textbf{return} \ i + 1 \\
\text{end}
\end{array}
\]

Running time is \(O(\quad )\)
Running Time of Partition($A, p, r$)

Partition($A, p, r$)

\begin{algorithm}

\begin{algorithmic}
\Statex \begin{align*}
\text{begin} \\
\quad x &= A[r]; \\
\quad i &= p - 1; \\
\quad \text{for } j = p \text{ to } r - 1 \text{ do} \\
\quad \quad \text{if } A[j] \leq x \text{ then} \\
\quad \quad \quad i &= i + 1; \\
\quad \quad \quad \text{exchange } A[i] \text{ and } A[j]; \quad \text{/* } O(r - p) \text{ */} \\
\quad \text{end} \\
\text{end} \\
\text{exchange } A[i + 1] \text{ and } A[r]; \\
\text{return } i + 1
\end{align*}
\end{algorithmic}
\end{algorithm}

Running time is $O(r - p)$
Running Time of Partition($A, p, r$)

Partition($A, p, r$)

begin
  x = A[r];
  i = p − 1;
  for $j = p$ to $r − 1$ do
    if $A[j] \leq x$ then
      i = i + 1;
      exchange $A[i]$ and $A[j]$; // $O(r − p)$
    end
  end
  exchange $A[i + 1]$ and $A[r]$;
return $i + 1$
end

Running time is $O(r − p)$
- linear in the length of the array $A[p..r]$
Running Time of Partition($A, p, r$)

Partition($A, p, r$)

begin
  $x = A[r]$;
  $i = p - 1$;
  for $j = p$ to $r - 1$ do
    if $A[j] \leq x$ then
      $i = i + 1$;
      exchange $A[i]$ and $A[j]$; // $O(r - p)$
    end
  end
  exchange $A[i + 1]$ and $A[r]$;
  return $i + 1$
end

Running time is $O(r - p)$

- linear in the length of the array $A[p..r]$
Quicksort

Quicksort(\(A, p, r\))

\[
\begin{align*}
\text{begin} & \quad \text{if} \ p < r \ \text{then} \\
& \quad q = \text{Partition}(A, p, r); \\
& \quad \text{Quicksort}(A, \ q); \\
& \quad \text{Quicksort}(A, \ r); \\
\text{end} & \quad \text{end}
\end{align*}
\]

If we could always partition the array into halves, then we have the recurrence

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + O(n),
\]

hence

\[
T(n) = O(n \log n).
\]

However, if we always get unlucky with very unbalanced partitions, then

\[
T(n) \leq T(n - 1) + O(n),
\]

hence

\[
T(n) = O(n^2).
\]
Quicksort

Quicksort($A, p, r$)

begin
  if $p < r$ then
    $q = \text{Partition}(A, p, r)$;
    Quicksort($A, p, q - 1$);
    Quicksort($A, q$, $r$);
  end
end

If we could always partition the array into halves, then we have the recurrence

$$T(n) \leq 2T(n/2) + O(n),$$

hence

$$T(n) = O(n \log n).$$

However, if we always get unlucky with very unbalanced partitions, then

$$T(n) \leq T(n - 1) + O(n),$$

hence

$$T(n) = O(n^2).$$
Quicksort

Quicksort(A, p, r)

begin
    if p < r then
        q = Partition(A, p, r);
        Quicksort(A, p, q - 1);
        Quicksort(A, q + 1, r);
    end
end

If we could always partition the array into halves, then we have the recurrence

\[ T(n) \leq 2T(n/2) + O(n), \]

hence

\[ T(n) = O(n \log n). \]

However, if we always get unlucky with very unbalanced partitions, then

\[ T(n) \leq T(n-1) + O(n), \]

hence

\[ T(n) = O(n^2). \]
If we could always partition the array into halves, then we have the recurrence $T(n) \leq 2T(n/2) + O(n)$, hence $T(n) = O(n \log n)$.

However, if we always get unlucky with very unbalanced partitions, then $T(n) \leq T(n - 1) + O(n)$, hence $T(n) = O(n^2)$. 

Quicksort($A, p, r$)

begin
  if $p < r$ then
    $q =$ Partition($A, p, r$);
    Quicksort($A, p, q - 1$);
    Quicksort($A, q + 1, r$);
  end
end
Outline:

- Partition
- Quicksort
- Average Case Analysis of Quicksort
Average Case Analysis of Quicksort

Measuring running time:
Measuring running time:

- The running time is dominated by the time spent in partition.
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) =$
Average Case Analysis of Quicksort

Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) = T(m) + \ldots$
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.
Recurrence: 

$$T(n) = T(m) + T(n - m - 1) + O(n)$$
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$

Worst Case:
Average Case Analysis of Quicksort

Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$

Worst Case:

$T(n) = T(0) + T(n - 1) + O(n)$
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$

Worst Case:

\begin{align*}
T(n) &= T(0) + T(n - 1) + O(n) \\
T(n) &= O(\quad)
\end{align*}
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$

Worst Case:

\[
T(n) = T(0) + T(n - 1) + O(n)
\]

\[
T(n) = O(n^2)
\]
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$

Worst Case:

$T(n) = T(0) + T(n - 1) + O(n)$

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

What inputs give worst case performance?
Measuring running time:

- The running time is dominated by the time spent in partition.
- The running time of the partition procedure can be measured by the number of key comparisons.
- Need to specify $m$, the size of the left partition block.

$T(n)$: running time on array of size $n$.

Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$

Worst Case:

$$
T(n) = T(0) + T(n - 1) + O(n) \\
T(n) = O(n^2)
$$

What inputs give worst case performance?

We will analyze average case running time.
Worst-case doesn’t make sense: for any given input, the worst case is very unlikely to happen.
1. Worst-case doesn’t make sense: for any given input, the worst case is very unlikely to happen

2. Use Average Case Analysis
Average Case Analysis

1. Worst-case doesn’t make sense: for any given input, the worst case is very unlikely to happen
2. Use Average Case Analysis
3. Assume every possible input permutation of the $n$ items are equally likely.
4. $n!$ permutations so each one has probability $\frac{1}{n!}$ of occurring
5. If $S_n$ is set of all permutations, $\sigma \in S_n$ is a possible input permutation, then average running time is

$$\frac{1}{n!} \sum_{\sigma \in S_n} C(\sigma)$$
Average Case Analysis

Let $A$ be the set of items in $A[p..r]$ and $\sigma$ a random permutation of $A$.

1. $A[r]$ is equally likely to be any item in $A$.
2. After running the partition algorithm on $A[p..r]$, the input to the new left and right subproblems are again random permutations (need to argue why).

Recall that if $X$ is a random variable and $E_1, E_2, \ldots, E_n$ are events that partition the probability space then we can write the expectation of $X$ in terms of the Expectation of $X$ conditioned on $E_i$. That is

$$E(X) = \sum_i E(X|E_i) \Pr(E_i).$$
Average Case Analysis

Assume that the input to is a random permutation of $N$ items.

- Let $C_N$ be the average amount of work performed on the input
- $C_0 = C_1 = 0$.
- Partition requires $N - 1$ comparisons
- Each item has probability $1/N$ of being pivot.
- If Item $k$ is pivot, the two remaining subproblems require $C_{k-1} + C_{N-k}$ average time

$$C_N = N - 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k})$$

$$= N - 1 + \frac{2}{N} \sum_{1 \leq k \leq N} C_{k-1}$$
Multiplying both sides of previous equation by $N$ and then rewriting the equation for $N - 1$ yields

$$NC_N = N(N - 1) + 2 \sum_{1 \leq k \leq N} C_{k - 1}, \quad (N - 1)C_{N-1} = (N - 1)(N - 2) + 2 \sum_{1 \leq k \leq N-1} C_{k - 1}.$$  

Subtracting the 2nd from the 1st and simplifying yields

$$NC_N = (N + 1)C_{N-1} + 2N - 2$$

Dividing both sides by $N(N + 1)$ gives

$$\frac{C_N}{N + 1} = \frac{C_{N-1}}{N} + \frac{2}{N + 1} - \frac{2}{N(N + 1)}.$$
Telescoping the recurrence down to $N = 3$ and recalling that $C_1 = 0$ yields

\[
\frac{C_N}{N + 1} = \frac{C_{N-1}}{N} + \frac{2}{N + 1} - \frac{2}{N(N + 1)}
\]

\[
= \frac{C_{N-2}}{N - 1} + \left( \frac{2}{N} - \frac{2}{(N - 1)N} \right) + \left( \frac{2}{N + 1} - \frac{2}{N(N + 1)} \right)
\]

\[
= \ldots
\]

\[
= \frac{C_1}{2} + 2 \sum_{i=3}^{N} \frac{1}{i + 1} - \sum_{i=3}^{N} \frac{2}{i(i + 1)}
\]

\[
= 2H_{N+1} - 2H_3 + O(1) = 2H_N + O(1)
\]

where $H_N = \sum_{i=1}^{N} \frac{1}{i}$ and we are using the fact that $\sum_{i=1}^{\infty} \frac{1}{i} \frac{1}{i}$ is bounded.
Average Case Analysis

We just saw that

\[ \frac{C_N}{N+1} = 2H_N + O(1). \]

\(H_N\) is called the \(N\)th Harmonic number and it is well known that

\[ H_n = \ln n + O(1). \]

So, we have just proven that the average number of operations performed running Quicksort on a random permutation of \(N\) items is

\[ C_N = 2(N+1)H_N + O(N) = 2N \ln N + O(N). \]
Quicksort is a divide and conquer algorithm.
The Quicksort code can be tuned
- When \( N \) is small, call Insertion Sort rather than Quicksort (on very small \( N \), Insertion sort is faster.
- Instead of using last item \( A[r] \) as pivot, set pivot to be median of first, last and middle item. (Why should this help?)

`qsort` under UNIX was an extremely popular sorting routine for decades. It was a finely tuned version of Quicksort

Quicksort was published by Tony Hoare in the Communications of the ACM 4(7), 1961.