Data Structures and Algorithms [1]

BITS-Pilani K. K. Birla Goa Campus

[1] Material for the presentation taken from Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Fourth Edition*;

- Average case running time : $O(n \lg n)$

- Average case running time : $O(n \lg n)$
- Constant factor associated with $O(n \lg n)$ is small compared to merge sort.

Ch. 7 : Quicksort

- Average case running time : $O(n \lg n)$
- Constant factor associated with $O(n \lg n)$ is small compared to merge sort. (Best practical choice.)

- Average case running time : $O(n \lg n)$
- Constant factor associated with $O(n \lg n)$ is small compared to merge sort. (Best practical choice.)
- Worst case running time : $O(n^2)$

Ch. 7 : Quicksort

- Average case running time : $O(n \lg n)$
- Constant factor associated with $O(n \lg n)$ is small compared to merge sort. (Best practical choice.)
- Worst case running time : $O(n^2)$
- Uses the divide-and-conquer paradigm

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

Quicksort : Partition

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quicksort : Partition

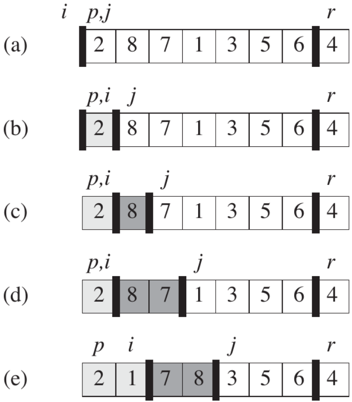
- A = [4, 2, 5, 1, 3] , p = 1 , r = 5

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$

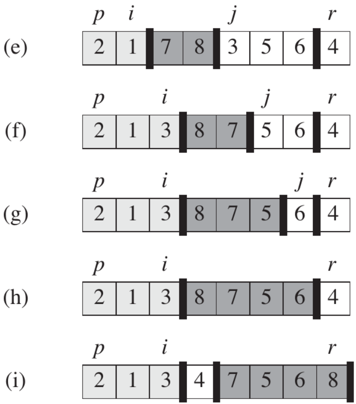
Quicksort : Partition

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

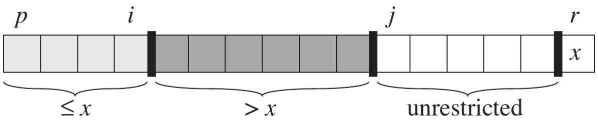


Quicksort : Partition

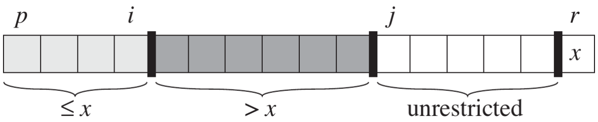
```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



PARTITION procedure: four regions

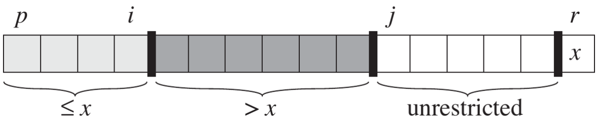


PARTITION procedure: four regions




```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

PARTITION procedure: four regions




```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Running time : $\Theta(n)$

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

$[\ 2,\ 5,\ 1,\ 4,\ \underline{3}\ ]$

Quicksort

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

[2, 5, 1, 4, $\underline{3}$]

[2, 1, $\underline{3}$, 4, 5]

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

[2, 5, 1, 4, $\underline{3}$]

[2, 1, $\underline{3}$, 4, 5]

[2, $\underline{1}$]   [3]   [4, 5]

Quicksort

$\textsc{Quicksort}(A, p, r)$

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

[2, 5, 1, 4, $\underline{3}$]

[2, 1, $\underline{3}$, 4, 5]

[2, $\underline{1}$] [3] [4, 5]

[$\underline{1}$, 2] [3] [4, 5]

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

$[\ 2,\ 5,\ 1,\ 4,\ \underline{3}\ ]$

$[\ 2,\ 1,\ \underline{3},\ 4,\ 5\ ]$

$[\ 2,\ \underline{1}\ ]\quad[\ 3\ ]\quad[\ 4,\ 5\ ]$

$[\ \underline{1},\ 2\ ]\quad[\ 3\ ]\quad[\ 4,\ 5\ ]$

$[\ 1\ ]\quad[\ 2\ ]\quad[\ 3\ ]\quad[\ 4,\ \underline{5}\ ]$

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

[2, 5, 1, 4, $\underline{3}$]

[2, 1, $\underline{3}$, 4, 5]

[2, $\underline{1}$] [3] [4, 5]

[$\underline{1}$, 2] [3] [4, 5]

[1] [2] [3] [4, $\underline{5}$]

[1] [2] [3] [4] [5]

Performance of quicksort

- Running time depends on whether the partition is balanced.

Performance of quicksort

- Running time depends on whether the partition is balanced.

  [2, 5, 9, 10, <u>13</u>]

Performance of quicksort

- Running time depends on whether the partition is balanced.

  [2, 5, 9, 10, $\underline{13}$]
- Worst-case partitioning

Performance of quicksort

- Running time depends on whether the partition is balanced.

  [2, 5, 9, 10, 13]
- Worst-case partitioning

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

Performance of quicksort

- Running time depends on whether the partition is balanced.

  [2, 5, 9, 10, $\underline{13}$]

- Worst-case partitioning

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Performance of quicksort

- Running time depends on whether the partition is balanced.
  [2, 5, 9, 10, $\underline{13}$]
- Worst-case partitioning

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

We can use substitution method : $T(n) = \Theta(n^2)$

Performance of quicksort

- Best-case partitioning

Performance of quicksort

- Best-case partitioning

$$T(n) = 2T(n/2) + \Theta(n)$$

Performance of quicksort

- Best-case partitioning

$$T(n) = 2T(n/2) + \Theta(n)$$

We can use master method : $T(n) = \Theta(n \lg n)$

Performance of quicksort

- Best-case partitioning

$$T(n) = 2T(n/2) + \Theta(n)$$

We can use master method : $T(n) = \Theta(n \lg n)$

- What would be the average-case performance?

Average-case performance

- Average case performance will be closer to the best-case performance

Average-case performance

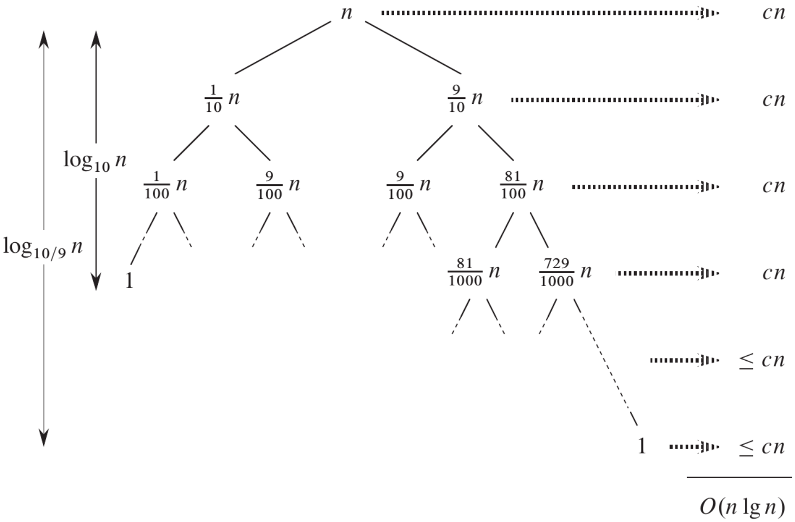
- Average case performance will be closer to the best-case performance
- Suppose 9-to-1 proportional split

Average-case performance

- Average case performance will be closer to the best-case performance
- Suppose 9-to-1 proportional split

$$T(n) = T(9n/10) + T(n/10) + cn$$

9-to-1 proportional split



Average-case performance

- Though a 9-to-1 split seems unbalanced, the running time is $O(n \lg n)$.

Average-case performance

- Though a 9-to-1 split seems unbalanced, the running time is $O(n \lg n)$.
- What if the split was 99-to-1? What would be the running time?

Average-case performance

- Though a 9-to-1 split seems unbalanced, the running time is $O(n \lg n)$.
- What if the split was 99-to-1? What would be the running time?
- Any split of the form rn and $(1-r)n$, where $0 < r < 1$ will give a running time of $O(n \lg n)$.

Average-case performance

- Though a 9-to-1 split seems unbalanced, the running time is $O(n \lg n)$.
- What if the split was 99-to-1? What would be the running time?
- Any split of the form rn and $(1-r)n$, where $0 < r < 1$ will give a running time of $O(n \lg n)$.
- It is unlikely that at each level PARTITION procedure will give us a highly unbalanced split (assuming a random input array).

Randomized version of quicksort

- We will uniformly randomly select an element to be the pivot element.

Randomized version of quicksort

- We will uniformly randomly select an element to be the pivot element.
- This should lead to the split produced by the PARTITION procedure to be well balanced on average.

Randomized Quicksort

```
RANDOMIZED-PARTITION(A, p, r)
    i = RANDOM(p, r)
    exchange A[r] with A[i]
    return PARTITION(A, p, r)
```

Randomized Quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Expected running time of randomized quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Expected running time of randomized quicksort

```
RANDOMIZED-PARTITION(A, p, r)
  i = RANDOM(p, r)
  exchange A[r] with A[i]
  return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

- How many times can the same element be selected as a pivot during the entire run of the quicksort algorithm?

Expected running time of randomized quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- How many times can the same element be selected as a pivot during the entire run of the quicksort algorithm?
- Therefore, the partition procedure will be called at most n times.

Expected running time of randomized quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

- How many times can the same element be selected as a pivot during the entire run of the quicksort algorithm?
- Therefore, the partition procedure will be called at most n times.
- So, the overall running time can be bounded by the number of times line 4 of the partition procedure is executed.

Expected running time of randomized quicksort

- *Lemma* : Let X be the number of comparisons performed in line 4 of the PARTITION procedure. Then the running time of quicksort is $O(n + X)$.

Expected running time of randomized quicksort

- *Lemma* : Let X be the number of comparisons performed in line 4 of the PARTITION procedure. Then the running time of quicksort is $O(n + X)$.
- We will derive an overall bound on the total number of comparisons X.

Expected running time of randomized quicksort

- *Lemma* : Let X be the number of comparisons performed in line 4 of the PARTITION procedure. Then the running time of quicksort is $O(n + X)$.
- We will derive an overall bound on the total number of comparisons X.
- We will rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element.

Expected running time of randomized quicksort

- *Lemma* : Let X be the number of comparisons performed in line 4 of the PARTITION procedure. Then the running time of quicksort is $O(n + X)$.
- We will derive an overall bound on the total number of comparisons X.
- We will rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element.

  $A = [5, 17, 13, 9, 15, 11, 3]$

Expected running time of randomized quicksort

- *Lemma* : Let X be the number of comparisons performed in line 4 of the PARTITION procedure. Then the running time of quicksort is $O(n+X)$.
- We will derive an overall bound on the total number of comparisons X.
- We will rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element.

  $A = [5, 17, 13, 9, 15, 11, 3]$

  What is z_1?

Expected running time of randomized quicksort

- *Lemma* : Let X be the number of comparisons performed in line 4 of the PARTITION procedure. Then the running time of quicksort is $O(n + X)$.
- We will derive an overall bound on the total number of comparisons X.
- We will rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element.

  $A = [5, 17, 13, 9, 15, 11, 3]$

  What is z_1? What is z_4?

Expected running time of randomized quicksort

- *Lemma* : Let X be the number of comparisons performed in line 4 of the PARTITION procedure. Then the running time of quicksort is $O(n + X)$.
- We will derive an overall bound on the total number of comparisons X.
- We will rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element.

  $A = [5, 17, 13, 9, 15, 11, 3]$

  What is z_1? What is z_4?
- Also, we will let Z_{ij} denote the set $\{z_i, z_{i+1}, \ldots, z_j\}$

Expected running time of randomized quicksort

- *Lemma* : Let X be the number of comparisons performed in line 4 of the PARTITION procedure. Then the running time of quicksort is $O(n + X)$.
- We will derive an overall bound on the total number of comparisons X.
- We will rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element.

  $A = [5, 17, 13, 9, 15, 11, 3]$

  What is z_1? What is z_4?
- Also, we will let Z_{ij} denote the set $\{z_i, z_{i+1}, \ldots, z_j\}$

  What is $Z_{3\,6}$?

Expected running time of randomized quicksort

- Lemma : Each pair of elements are compared at most once.

Expected running time of randomized quicksort

- Lemma : Each pair of elements are compared at most once.
  $A = [5, 17, 13, 9, 15, 11, 3]$

Expected running time of randomized quicksort

- Lemma : Each pair of elements are compared at most once.

  $A = [5, 17, 13, 9, 15, 11, 3]$

- Indicator random variable:

  $X_{ij} = I \qquad \{z_i \text{ is compared to } z_j\}$

Expected running time of randomized quicksort

- Lemma : Each pair of elements are compared at most once.

  $A = [5, 17, 13, 9, 15, 11, 3]$
- Indicator random variable:

  $X_{ij} = I \qquad \{z_i \text{ is compared to } z_j\}$
- Total number of comparisons X:

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

Expected number of comparisons

- Expected value of the total number of comparisons $E[X]$:

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

Expected number of comparisons

- Expected value of the total number of comparisons $E[X]$:

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right]
\end{aligned}
$$

Expected number of comparisons

- Expected value of the total number of comparisons $E[X]$:

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\}
\end{aligned}
$$

Expected number of comparisons

- RANDOMIZED-PARTITION procedure chooses each pivot uniformly randomly.

Expected number of comparisons

- RANDOMIZED-PARTITION procedure chooses each pivot uniformly randomly.
- Let us think: when will two elements **not** be compared?

Expected number of comparisons

- RANDOMIZED-PARTITION procedure chooses each pivot uniformly randomly.
- Let us think: when will two elements **not** be compared?

  $A = [5, 17, 13, 9, 15, 11, 3]$

Expected number of comparisons

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

Expected number of comparisons

- Once pivot x is chosen such that $z_i < x < z_j$, we know that z_i and z_j cannot be compared at any subsequent time.
- On the other hand, if z_i is chosen as a pivot before any other element in Z_{ij}, then z_i will be compared with all the elements in Z_{ij}.

Expected number of comparisons

- Once pivot x is chosen such that $z_i < x < z_j$, we know that z_i and z_j cannot be compared at any subsequent time.
- On the other hand, if z_i is chosen as a pivot before any other element in Z_{ij}, then z_i will be compared with all the elements in Z_{ij}.
- Similarly, if z_j is chosen as a pivot before any other element in Z_{ij}, then z_j will be compared with all the elements in Z_{ij}.

Expected number of comparisons

- In any run of the randomized quicksort algorithm, all elements in the set Z_{ij} are equally likely to be chosen as the first pivot from the set.

Probability that z_i is compared to z_j

$$\Pr\{z_i \text{ is compared to } z_j\} = \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}$$

Probability that z_i is compared to z_j

$$
\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\}
\end{aligned}
$$

Probability that z_i is compared to z_j

$$\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1}
\end{aligned}$$

Probability that z_i is compared to z_j

$$\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} .
\end{aligned}$$

Expected number of comparisons

Expected value of the total number of comparisons $E[X]$:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\}$$

Expected number of comparisons

Expected value of the total number of comparisons $E[X]$:

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}
\end{aligned}
$$

Expected number of comparisons

Expected value of the total number of comparisons $E[X]$:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\}$$

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

Expected number of comparisons

Expected value of the total number of comparisons $E[X]$:

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}
\end{aligned}
$$

Expected number of comparisons

Expected value of the total number of comparisons $E[X]$:

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \qquad \left(\text{Using } \sum_{k=1}^{n} \frac{1}{k} = O(\lg n) \right)
\end{aligned}
$$

Expected number of comparisons

Expected value of the total number of comparisons $E[X]$:

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \qquad \left(\text{ Using } \sum_{k=1}^{n} \frac{1}{k} = O(\lg n)\ \right) \\
&= O(n \lg n)
\end{aligned}
$$

Expected running time of randomized quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Expected running time $= O(n + X) =$

Expected running time of randomized quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Expected running time $= O(n + X) = O(n + cn\lg n) =$

Expected running time of randomized quicksort

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Expected running time $= O(n + X) = O(n + cn \lg n) = O(n \lg n)$

- Comparison sort

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Quicksort Partition procedure

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Comparison sort

- Worst case running time of Insertion sort $= \Theta(n^2) =$

Comparison sort

- Worst case running time of Insertion sort $= \Theta(n^2) = \Omega(n \lg n)$

Comparison sort

- Worst case running time of Insertion sort $= \Theta(n^2) = \Omega(n \lg n)$
- Worst case running time of Merge sort and Heap sort $=$

Comparison sort

- Worst case running time of Insertion sort $= \Theta(n^2) = \Omega(n \lg n)$
- Worst case running time of Merge sort and Heap sort $=$ $\Theta(n \lg n) =$

Comparison sort

- Worst case running time of Insertion sort $= \Theta(n^2) = \Omega(n \lg n)$
- Worst case running time of Merge sort and Heap sort $= \Theta(n \lg n) = \Omega(n \lg n)$

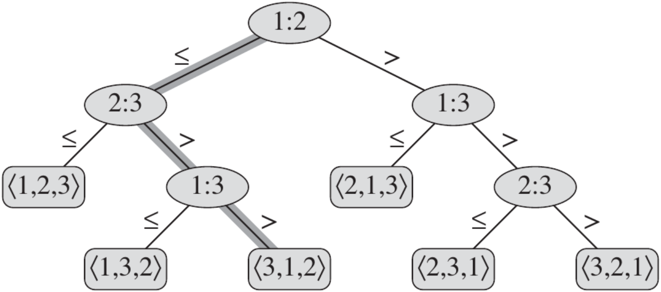
Comparison sort

- Worst case running time of Insertion sort $= \Theta(n^2) = \Omega(n \lg n)$
- Worst case running time of Merge sort and Heap sort $= \Theta(n \lg n) = \Omega(n \lg n)$
- Is it possible to have a better worst case running time?

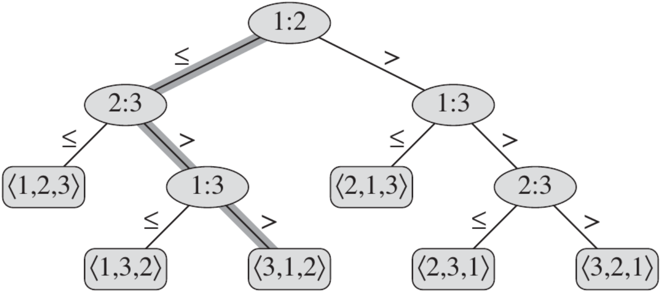
Comparison sort

- Worst case running time of Insertion sort $= \Theta(n^2) = \Omega(n \lg n)$
- Worst case running time of Merge sort and Heap sort $= \Theta(n \lg n) = \Omega(n \lg n)$
- Is it possible to have a better worst case running time?
- Can we have a sorting algorithm whose worst case running time is $o(n \lg n)$?

Decision tree for comparison sort

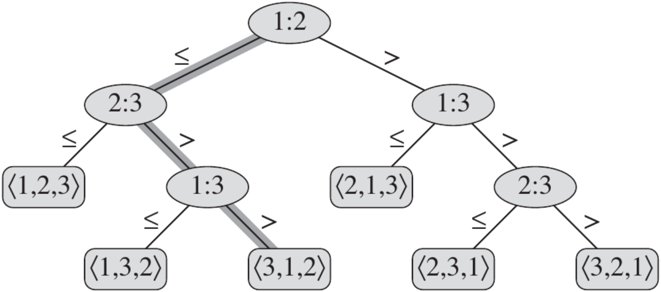


Decision tree for comparison sort



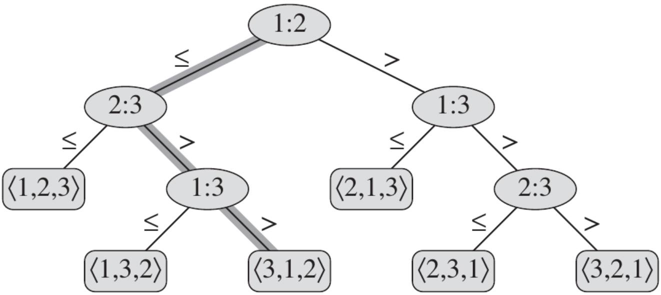
- In the worst case, we must perform at least h comparison operations.

Decision tree for comparison sort



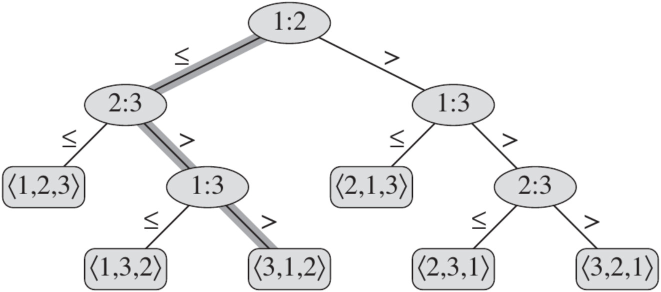
- In the worst case, we must perform at least h comparison operations.
- The height of the decision tree gives a lower bound on the number of comparisons needed in the worst-case.

Decision tree for comparison sort

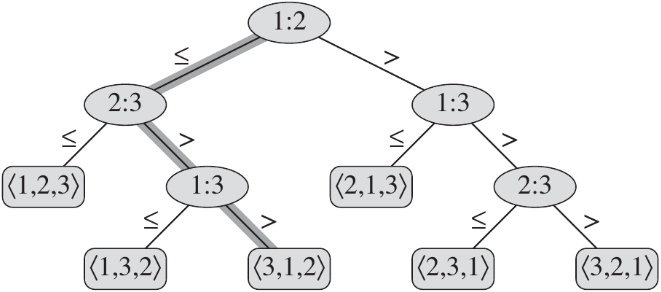


- In the worst case, we must perform at least h comparison operations.
- The height of the decision tree gives a lower bound on the number of comparisons needed in the worst-case.
- Any correct sorting algorithm must be able to produce each of the $n!$ permutations of the n input elements.

Decision tree is a full binary tree

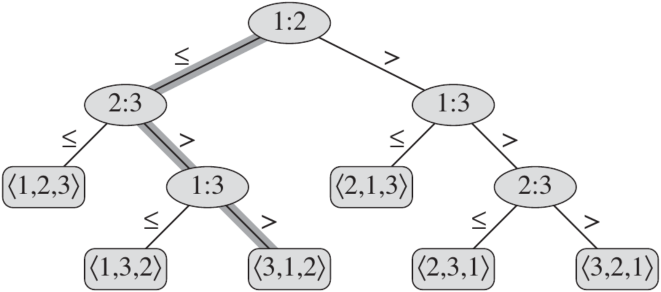


Decision tree is a full binary tree



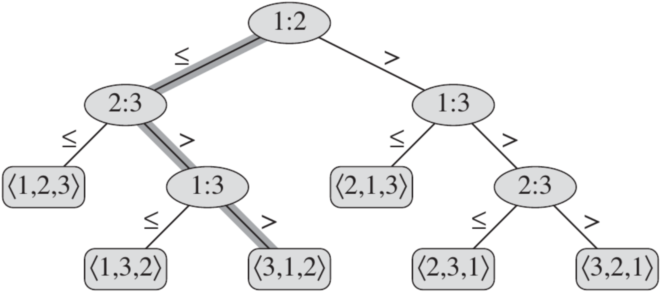
- A **full binary tree** is a binary tree in which every node has either 0 or 2 child nodes.

Decision tree is a full binary tree



- A **full binary tree** is a binary tree in which every node has either 0 or 2 child nodes.
- Maximum number of leaf nodes in a full binary tree of height $h =$

Decision tree is a full binary tree



- A **full binary tree** is a binary tree in which every node has either 0 or 2 child nodes.
- Maximum number of leaf nodes in a full binary tree of height $h = 2^h$

Lower bound on the number of comparisons

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

Lower bound on the number of comparisons

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

  Let the decision tree corresponding to the sorting algorithm be of height h and have l leaf nodes.

Lower bound on the number of comparisons

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

  Let the decision tree corresponding to the sorting algorithm be of height h and have l leaf nodes.

  $$n! \leq l$$

Lower bound on the number of comparisons

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

  Let the decision tree corresponding to the sorting algorithm be of height h and have l leaf nodes.

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

Lower bound on the number of comparisons

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

  Let the decision tree corresponding to the sorting algorithm be of height h and have l leaf nodes.

$$n! \leq l \leq 2^h$$
$$h \geq \lg(n!)$$

Lower bound on the number of comparisons

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

  Let the decision tree corresponding to the sorting algorithm be of height h and have l leaf nodes.

$$n! \leq l \leq 2^h$$
$$h \geq \lg(n!) = \Theta(n \lg n)$$

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

  Let the decision tree corresponding to the sorting algorithm be of height h and have l leaf nodes.

$$n! \leq l \leq 2^h$$
$$h \geq \lg(n!) = \Theta(n \lg n) \text{ (Stirling's approximation)}$$

Lower bound on the number of comparisons

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

  Let the decision tree corresponding to the sorting algorithm be of height h and have l leaf nodes.

$$
\begin{aligned}
n! &\leq l \leq 2^h \\
h &\geq \lg(n!) = \Theta(n \lg n) \text{ (Stirling's approximation)} \\
&= \Omega(n \lg n)
\end{aligned}
$$

Order of growth of $\lg(n!)$

- Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

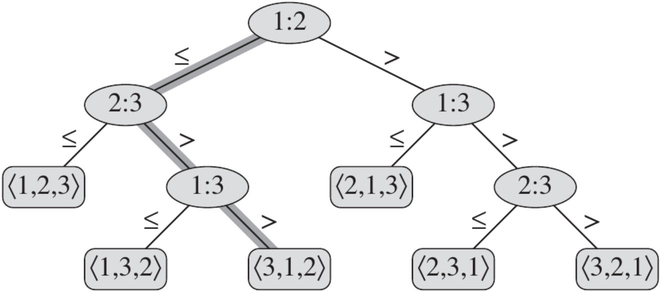
Order of growth of $\lg(n!)$

- Stirling's approximation:
$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$
- Leading term in Stirling's approximation is $n^{n+\frac{1}{2}}$

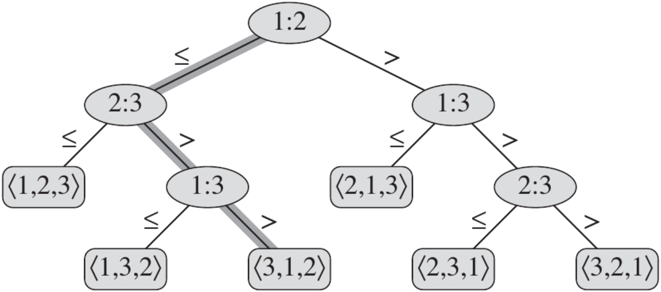
Order of growth of $\lg(n!)$

- Stirling's approximation:
$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)$$
- Leading term in Stirling's approximation is $n^{n+\frac{1}{2}}$
- $\lg(n!) = \Theta(n \lg n)$

$h = \Omega(n \lg n)$

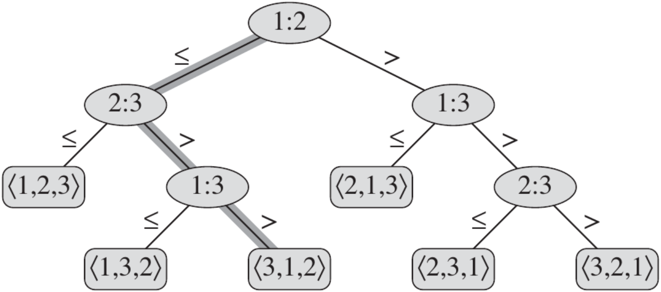


$h = \Omega(n \lg n)$



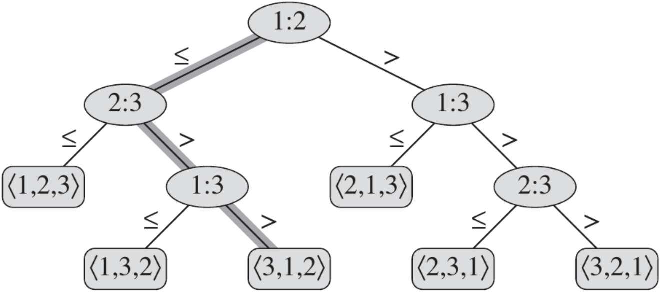
- Any comparison sort algorithm would require $\Omega(n \lg n)$ comparisons in the worst case.

$h = \Omega(n \lg n)$



- Any comparison sort algorithm would require $\Omega(n \lg n)$ comparisons in the worst case.
- Any comparison based algorithm would take at least $\Omega(n \lg n)$ time in the worst case for solving the sorting problem.

$h = \Omega(n \lg n)$



- Any comparison sort algorithm would require $\Omega(n \lg n)$ comparisons in the worst case.
- Any comparison based algorithm would take at least $\Omega(n \lg n)$ time in the worst case for solving the sorting problem.
- Heapsort and Merge sort are asymptotically optimal algorithm for comparison sorting.

Counting sort

- Assumption : Each input element is an *integer* in the range 0 to k for some integer k.

Counting sort

- Assumption : Each input element is an *integer* in the range 0 to k for some integer k.
- Input array : $A[1 \dots n]$

Counting sort

- Assumption : Each input element is an *integer* in the range 0 to k for some integer k.
- Input array : $A[1 \dots n]$
- Output array : $B[1 \dots n]$

Counting sort

- Assumption : Each input element is an *integer* in the range 0 to k for some integer k.
- Input array : $A[1 \dots n]$
- Output array : $B[1 \dots n]$
- Temporary working storage : $C[0 \dots k]$

Counting sort operations

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

Counting sort operations

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

Counting sort operations

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

(c)

Counting sort operations

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

(c)

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

(d)

Counting sort operations

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

(a)

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

Counting sort operations

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

(a)

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

Counting sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

Running time of Counting sort

- Overall running time $\Theta(k + n)$

Running time of Counting sort

- Overall running time $\Theta(k + n)$
- When $k = O(n)$, the counting sort runs in $\Theta(n)$ time.

Running time of Counting sort

- Overall running time $\Theta(k + n)$
- When $k = O(n)$, the counting sort runs in $\Theta(n)$ time.
- Why running time of Counting sort has a better worst case lower bound?

Running time of Counting sort

- Overall running time $\Theta(k + n)$
- When $k = O(n)$, the counting sort runs in $\Theta(n)$ time.
- Why running time of Counting sort has a better worst case lower bound?
- Counting sort is **stable**. (What is a stable sorting algorithm?)

Is Insertion sort stable?

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Is Quick sort stable?

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Stability of sorting algorithms
  - Merge sort

- Stability of sorting algorithms
  - Merge sort
  - Heap sort

- Stability of sorting algorithms
  - Merge sort
  - Heap sort
- Sorting in linear time

- Stability of sorting algorithms
  - Merge sort
  - Heap sort
- Sorting in linear time
  - Radix sort

- Stability of sorting algorithms
  - Merge sort
  - Heap sort
- Sorting in linear time
  - Radix sort
  - Bucket sort