CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #5

➢ Quicksort

Prof. Pietro S. Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

Reading: Chapter 7

Aims of this lecture

- To introduce the **QuickSort** algorithm: a popular algorithm which is fast in practice, despite a $\Theta(n^2)$ worst case time.
- To show an **average-case analysis**, revealing why QuickSort is fast in practice.
- To see another example of **divide-and-conquer.**

➢ Idea behind QuickSort

- **Divide**:
  - Pick some element called **pivot**.
  - Move it to its final location in the sorted sequence such that **all smaller elements** are to its **left**, **larger** ones are to its **right.**
- **Conquer**:
  - Recursively sort subarrays for smaller and larger elements
- **Combine**:
  - No work needed here – after the recursion the array is sorted.

QuickSort: The Algorithm

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

if $p < r$ **then**
 $q = \textsc{Partition}(A, p, r)$
 $\textsc{QuickSort}(A, p, q - 1)$
 $\textsc{QuickSort}(A, q + 1, r)$

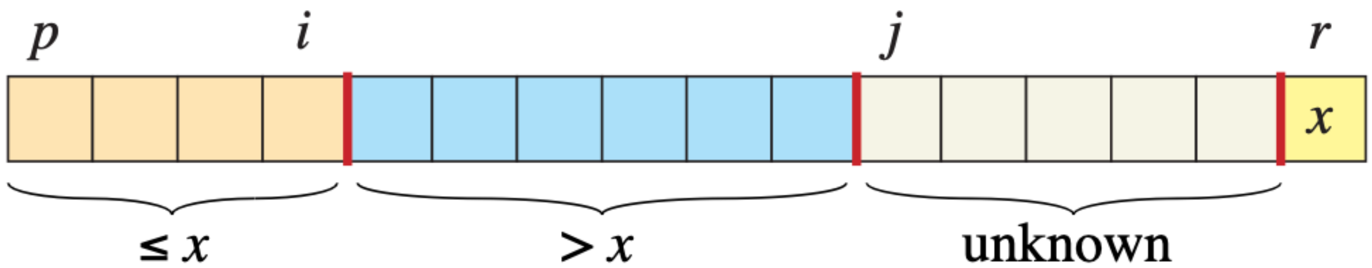
Initial call: $\textsc{QuickSort}(A, 1, \text{A.length})$

Differences to MergeSort:

- Split the array at *q*, the position of the pivot in sorted array
  - We don't know *q* in advance, it is revealed by Partition
- No combine step at the end
- Partition plays a similar role to Merge

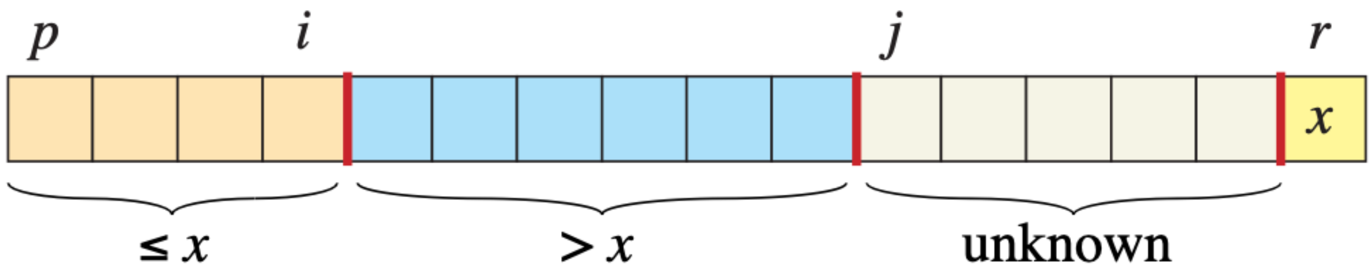
➢ **Partition**(A, p, r)

- Rearranges the subarray $A[p..r]$ in place, using swaps
- Takes the last element $A[r]$ as pivot element.
- Idea:
  - Scan the subarray from left to right
  - Build up a subarray $A[p..i]$ of elements smaller or equal to the pivot
  - Build up a subarray $A[i+1..j-1]$ of elements larger than the pivot
  - When reaching the end of the array, put the pivot in the right place



Partition: Pseudocode

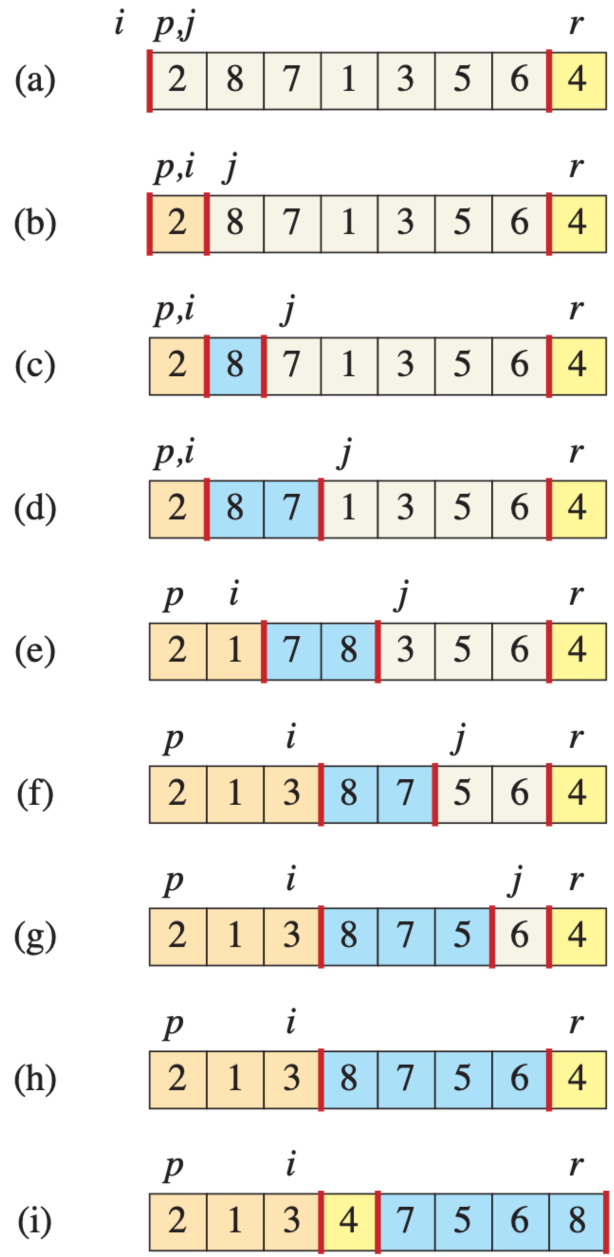
```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1 do
        if A[j] ≤ x then
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



Partition: Example

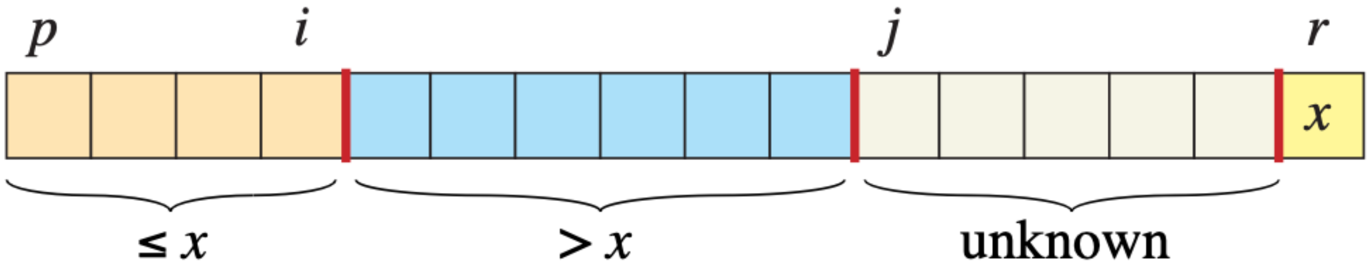
PARTITION(A, p, r)

```
1: x = A[r]
2: i = p − 1
3: for j = p to r − 1 do
4:     if A[j] ≤ x then
5:         i = i + 1
6:         exchange A[i] with A[j]
7: exchange A[i + 1] with A[r]
8: return i + 1
```



(step (i) swaps pivot into place, line 7)

Partition: Correctness (1)




```
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] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Loop invariant:
At the beginning of the j_th iteration:

$$A[p]..A[i] \leq x$$

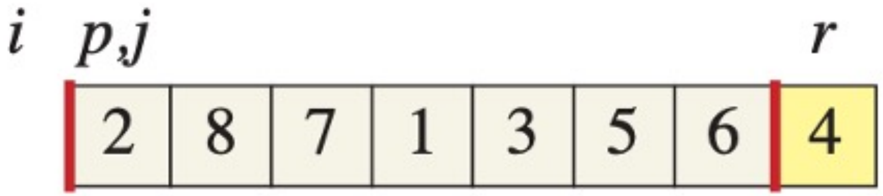
and

$$A[i+1]..A[j-1] > x.$$

- See picture above –

➢ Partition: Initialisation

(a)



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

```
x = A[r]
i = p − 1
for j = p to r − 1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Loop invariant:
See picture above –

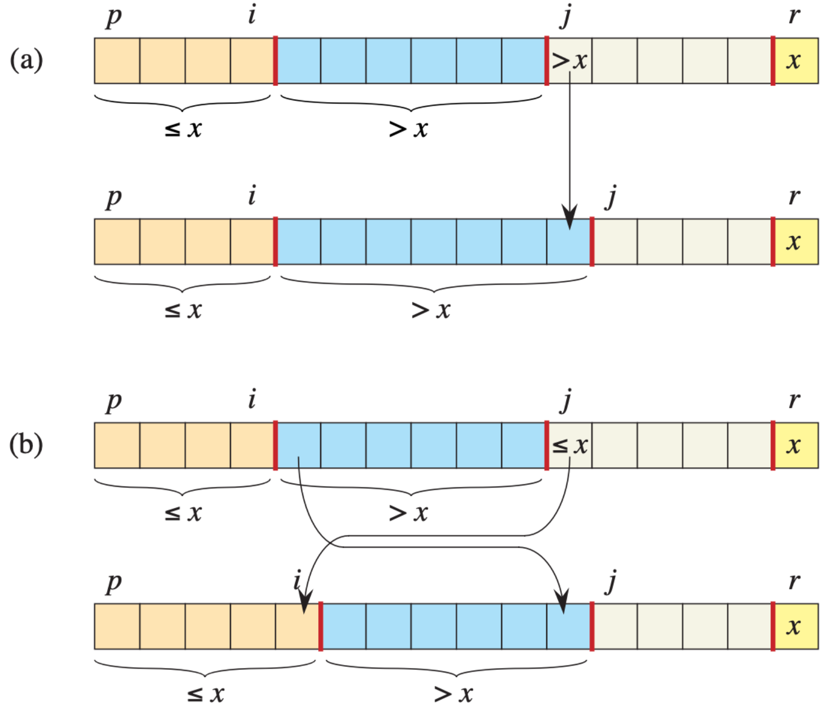
$$A[p]..A[i] \leq x$$
and
$$A[i+1]..A[j-1] > x.$$

Trivially true at initialisation.
(both sets are empty)

Partition: Maintaining the loop invariant

PARTITION(A, p, r)

1: $x = A[r]$
2: $i = p - 1$
3: **for** $j = p$ to $r - 1$ **do**
4:     **if** $A[j] \leq x$ **then**
5:         $i = i + 1$
6:         exchange $A[i]$ with $A[j]$
7: exchange $A[i+1]$ with $A[r]$
8: **return** $i + 1$



Maintenance:

- If line 4 is false : picture (a)
- If line 4 true: picture (b)

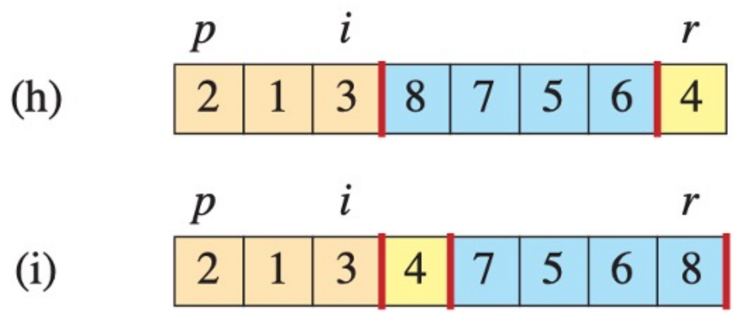
- In both cases after one iteration of *j* the loop invariant is maintained.

Loop invariant:

$$A[p]..A[i] \leq x$$
and
$$A[i+1]..A[j-1] > x.$$

Partition: termination

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



Loop invariant:

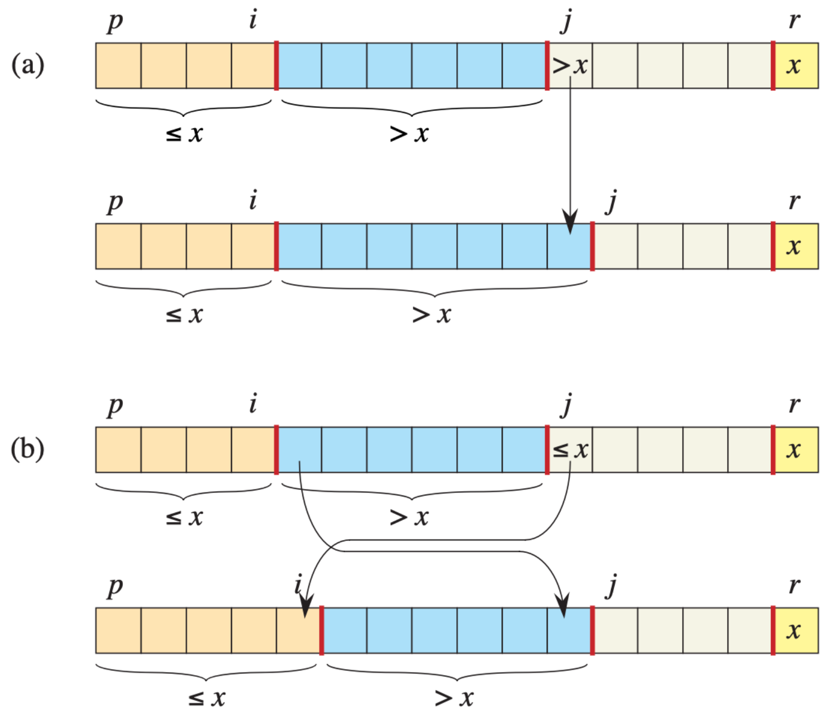
$$A[p]..A[i] \leq x$$
and
$$A[i+1]..A[j-1] > x.$$

Termination:
After the last swap in line 7,
$A[p]..A[i] \leq x < A[i+2]..A[r]$
and Partition returns the position of x.

Exercise: Analyse the Runtime of Partition

Q: What is the runtime of Partition on a subarray of size n?

```
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] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



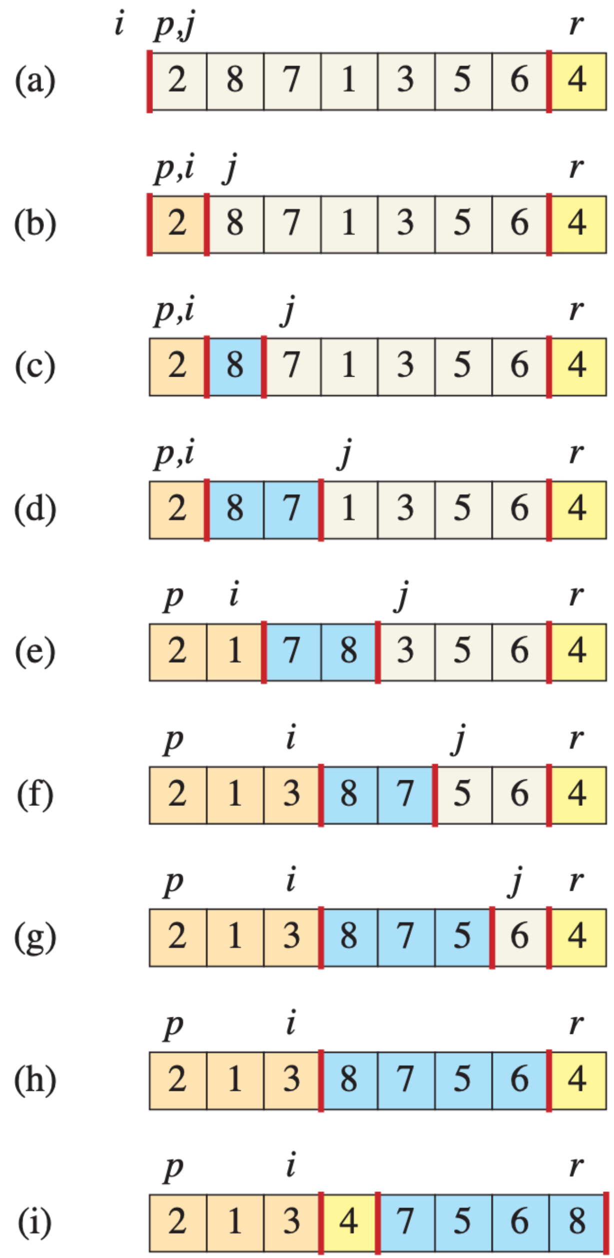
QuickSort: The Algorithm

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

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

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

```
x = A[r]
i = p − 1
for j = p to r − 1 do
    if A[j] ≤ x then
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



Runtime?

Worst-case and Best-case Partitionings

- The overall runtime depends on **how the array is partitioned** as that determines the sizes $q-1$ and $r-q$ of the subarray to be sorted recursively.
  - Recall that we don't know in advance where the pivot will end up.
- **Questions**:
  - What might be a **worst-case partitioning** for the runtime?
  - What might be a **best-case partitioning** for the runtime?

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

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

Worst-case Partitioning

- The worst case is attained when Partition always produces one subproblem with $n - 1$ and one with 0 elements.
- This is the case, for example, when the array is already sorted.
- This leads to the following recurrence:

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

- Solving this gives $T(n) = \Theta(n^2)$.

Best-case Partitioning

- Best case: split into two subproblems of sizes $\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lceil \frac{n}{2} \right\rceil - 1$.
- Ignoring floors, ceilings, and -1 we get the recurrence:

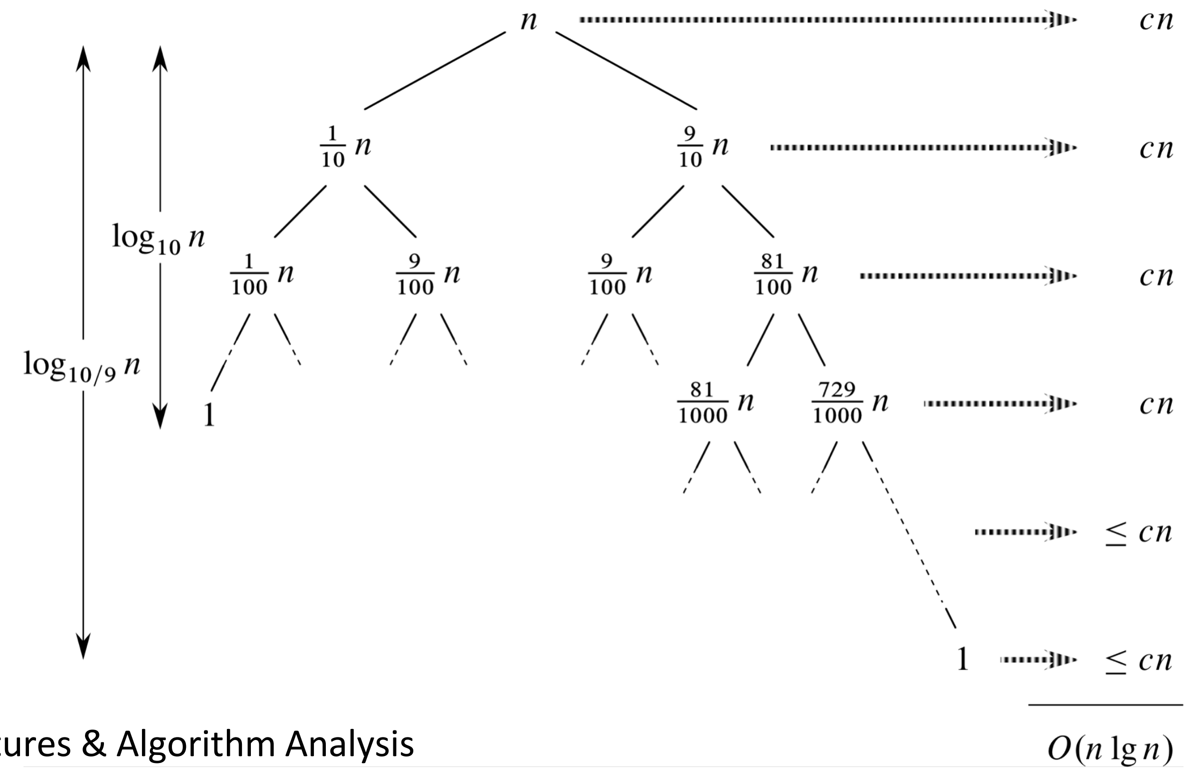
$$T(n) = 2T(n/2) + \Theta(n)$$

- Deja vu?
- This is $\Theta(n \log n)$ from the analysis of MergeSort.
- True to the spirit of divide-and-conquer.

Towards an average case

- What if the split was always $\frac{9}{10} \cdot n$ and $\frac{1}{10} \cdot n$?
- Getting the recurrence

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



Average case analysis

- Assume all elements of the array are distinct.
- Assume each split $q = 1, 2, \ldots, n$ was equally likely.
- This situation occurs when the input is chosen **uniformly at random** amongst all $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ possible orderings.
- Then

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

- Average over all problem sizes for 2 subproblems $+\Theta(n)$.
- Solving this recurrence gives a bound of $O(n \log n)$.
- We prove this next!

Average case analysis (2): Substitution method

To prove: $$\frac{1}{n} \cdot \sum_{k=0}^{n-1} 2T(k) + \Theta(n) \le \boldsymbol{c\, n \ln n}$$

Base case: n=2 Prove: $T(2) \le c2\ln 2$

$$\mathrm{T}(2) = \mathrm{T}(0) + \mathrm{T}(1) + \Theta(2) = 2\mathrm{c}' + \mathrm{c}^* \le \mathrm{c}\ 2\ln 2 \quad (for \quad e.g., c > 2c' + \mathrm{c}^*)$$

Inductive case: Assume true for **<n** **(**$T(k) \le c\ k\ \ln k$ for $k\ < n$) and prove for **n**

$$\mathrm{T(n)} = \frac{2}{n}\left[\mathrm{T}(0) + \mathrm{T}(1) + \sum_{k=2}^{n-1} c\ k\ \ln k\right] + \Theta(n)$$

$$\le \frac{2}{n}\left[\mathrm{c}' + \mathrm{c}' + \sum_{k=2}^{n-1} c\ k\ \ln k\right] + \Theta(n) =$$

$$= \frac{4c'}{n} + \left(\frac{2c}{n}\ \sum_{k=2}^{n-1} k \ln k\right) + \Theta(n) \le \frac{4c'}{n} + \frac{2c}{n}\left[\frac{n^2\ \ln n}{2} - \frac{n^2}{4}\right] + \Theta(n)$$

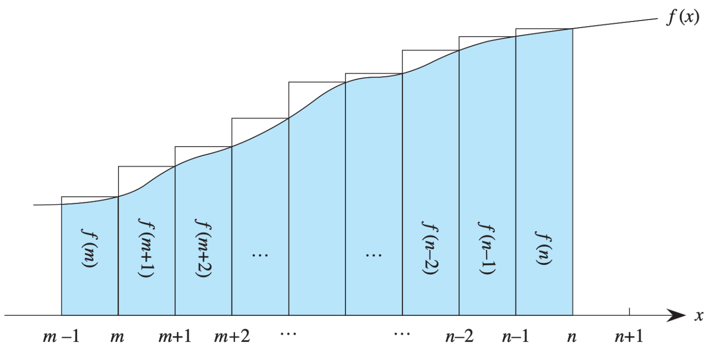
$$= \mathrm{c}\, n \ln n - \frac{cn}{2} + \frac{4c'}{n} + \ \Theta(n) < c\,\mathrm{n}\ln n \iff \frac{cn}{2} > \frac{4c'}{n} + c^*\, n,\ (e.g. for\ c > 3c^*)$$

➢ Average case analysis (3)

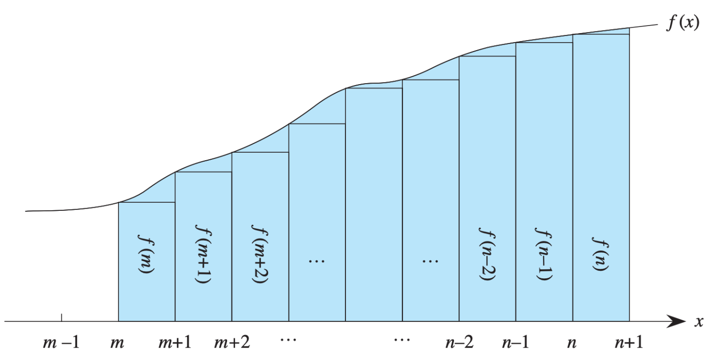
$$\sum_{k=2}^{n-1} k \ln k \le \int_{2}^{n} k \ln k \; dk = \left[\frac{n^2 \ln n}{2} - \frac{n^2}{4}\right]$$

When a summation has the form $\sum_{k=m}^{n} f(k)$, where $f(k)$ is a monotonically increasing function, you can approximate it by integrals:

$$\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx \;.$$



(a)



➢ Improvements to QuickSort

- QuickSort is fast in practice because of small constants in the asymptotic running time.
- Improvements for handling **equal values** (exercise)
  - Partition into smaller, equal and larger elements
  - Only need to sort smaller and larger subarrays
- Choose the pivot as **median of 3** elements
  - Slightly faster in practice, but still quadratic worst case
- **Dual-Pivot QuickSort** by Vladimir Yaroslavskiy
  - Use two pivots instead of one and partition array in 3 areas
  - Used in Java 7

A Randomised Version of QuickSort

- Choosing the right pivot element can be tricky – we have no idea *a priori* which pivot elements are good.
- **Solution**: **leave it to chance!**

$\textsc{Randomised-Partition}(A, p, r)$

$i = \textsc{Random}(p, r)$
exchange $A[r]$ with $A[i]$
return $\textsc{Partition}(A, p, r)$

$\textsc{Randomised-QuickSort}(A, p, r)$

if $p < r$ **then**
    $q = \textsc{Randomised-Partition}(A, p, r)$
    $\textsc{Randomised-QuickSort}(A, p, q-1)$
    $\textsc{Randomised-QuickSort}(A, q+1, r)$

"Random" picks pivot uniformly at random among all elements.

Summary

- QuickSort is used in modern programming languages
  - QuickSort has a bad worst-case runtime of $\Theta(n^2)$
  - Average-case performance on **random inputs** is $O(n \log n)$.
- Why is it popular?
  - Constants hidden in the asymptotic terms are small.
- Next week we'll see how randomisation allows to avoid the worst case runtime