Introduction to Algorithms

Topic 3 : Comparison Based Sorting Algorithms

Xiang-Yang Li and Haisheng Tan[1]

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2025

Outline

Basic Concepts of Sorting Algorithm

Stability

Regardless of how the input data is distributed, the data objects of the same keyword will be kept in the same order as in the input during the sorting process, which is called stable sorting. Otherwise, called unstable sorting.

Example: $2, 2^*, 1 \rightarrow 1, 2^*, 2$ (unstable sorting)

Basic Concepts of Sorting Algorithm

Stability

Regardless of how the input data is distributed, the data objects of the same keyword will be kept in the same order as in the input during the sorting process, which is called stable sorting. Otherwise, called unstable sorting.

Example: $2, 2^*, 1 \rightarrow 1, 2^*, 2$ (unstable sorting)

Time Complexity

Usually measured by the number of data comparisons and the number of data movements in the algorithm execution.

Basic Concepts of Sorting Algorithm

Stability

Regardless of how the input data is distributed, the data objects of the same keyword will be kept in the same order as in the input during the sorting process, which is called stable sorting. Otherwise, called unstable sorting.

Example: $2, 2^*, 1 \rightarrow 1, 2^*, 2$ (unstable sorting)

Time Complexity

Usually measured by the number of data comparisons and the number of data movements in the algorithm execution.

In-place Sorting

only a constant of elements are stored outside the input array.

Contents

Insertion Sort

General idea: Maintain an ordered sequence.

Insertion Sort

General idea: Maintain an ordered sequence.

```
Insertion-Sort(A)
for j = 2 to A.length do
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j − 1].
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Example of Insertion Sort

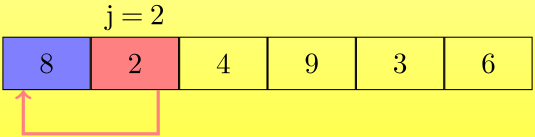
8	2	4	9	3	6

Example of Insertion Sort

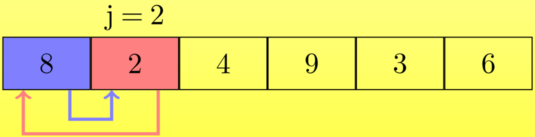
$j = 2$

8	2	4	9	3	6

Example of Insertion Sort



Example of Insertion Sort



Example of Insertion Sort

j = 3

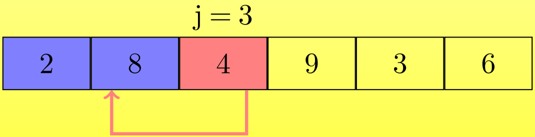
2	8	4	9	3	6

Example of Insertion Sort

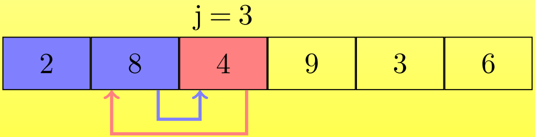
j = 3

2	8	4	9	3	6

Example of Insertion Sort



Example of Insertion Sort



Example of Insertion Sort

j = 4

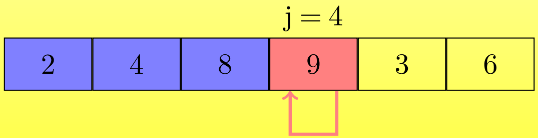
2	4	8	9	3	6

Example of Insertion Sort

$j = 4$

2	4	8	9	3	6

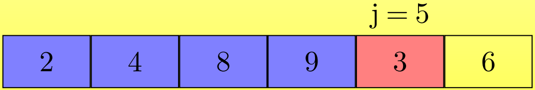
Example of Insertion Sort



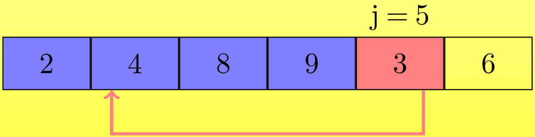
Example of Insertion Sort

				j = 5	
2	4	8	9	3	6

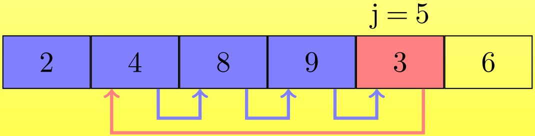
Example of Insertion Sort



Example of Insertion Sort



Example of Insertion Sort

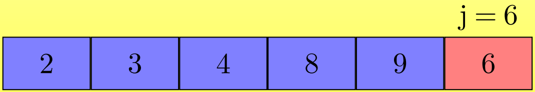


Example of Insertion Sort

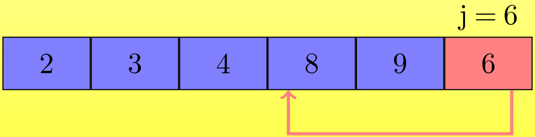
$j = 6$

2	3	4	8	9	6

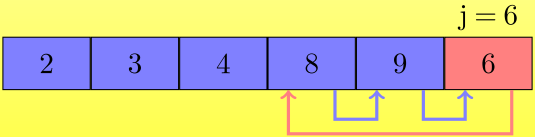
Example of Insertion Sort



Example of Insertion Sort



Example of Insertion Sort



Example of Insertion Sort

2	3	4	6	8	9

Insertion Sort

- Time Complexity
  - Best: $O(n)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- Memory: 1
- Stable: Yes

Insertion Sort

- Time Complexity
  - Best: $O(n)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- Memory: 1
- Stable: Yes

Insertion-Sort(A)

```
for j = 2 to A.length do
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j − 1].
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Contents

Selection Sort

General idea: Select and remove the smallest element from unsorted set.

Selection Sort

General idea: Select and remove the smallest element from unsorted set.

```
Selection-Sort(A)
for i = 1 to A.length − 1 do
    k = i                        ▷ k is the position of the smallest key.
    for j = i + 1 to A.length do
        if A[j] < A[k] then
            k = j
    if k ≠ i then
        A[i] ↔ A[k]
```

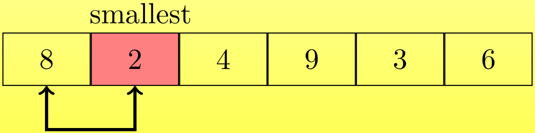
Example of Selection Sort

8	2	4	9	3	6

Example of Selection Sort

	smallest				
8	2	4	9	3	6

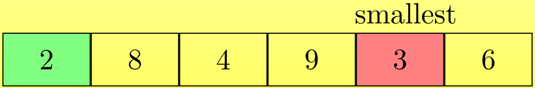
Example of Selection Sort



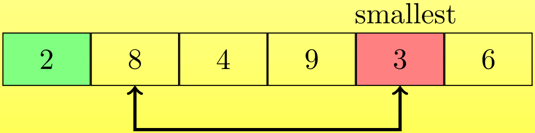
Example of Selection Sort

2	8	4	9	3	6

Example of Selection Sort



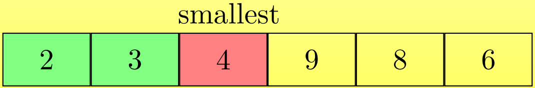
Example of Selection Sort



Example of Selection Sort

2	3	4	9	8	6

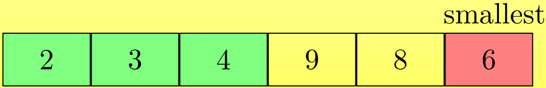
Example of Selection Sort



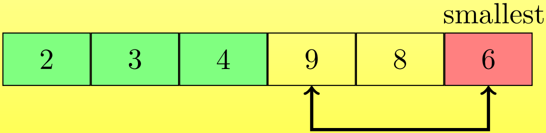
Example of Selection Sort

2	3	4	9	8	6

Example of Selection Sort



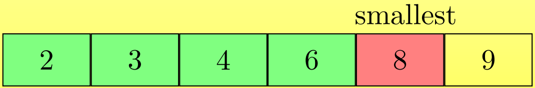
Example of Selection Sort



Example of Selection Sort

2	3	4	6	8	9

Example of Selection Sort



Example of Selection Sort

2	3	4	6	8	9

Selection Sort

- Time Complexity
  - Best: $O(n^2)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- Memory: 1
- Stable: No

Selection Sort

- Time Complexity
  - Best: $O(n^2)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- Memory: 1
- Stable: No

```
Selection-Sort(A)
for i = 1 to A.length − 1 do
    k = i
    for j = i + 1 to A.length do
        if A[j] < A[k] then
            k = j
    if k ≠ i then
        A[i] ↔ A[k]
```

Selection Sort

Stable sorting: How to revise the selection sorting to make it stable?

Contents

Bubble Sort

General idea: From the back to the front, if some elements are smaller than their predecessor, then swap them.

Bubble Sort

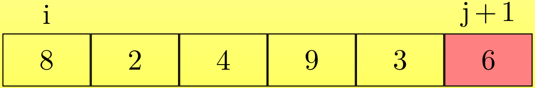
General idea: From the back to the front, if some elements are smaller than their predecessor, then swap them.

```
Bubble-Sort(A)
for i = 1 to A.length − 1 do
    noswap = TRUE
    for j = A.length − 1 downto i do
        if A[j + 1] < A[j] then
            A[j] ↔ A[j + 1]
            noswap = FALSE
    if noswap then break
```

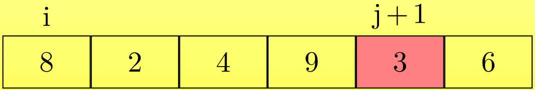
Example of Bubble Sort

8	2	4	9	3	6

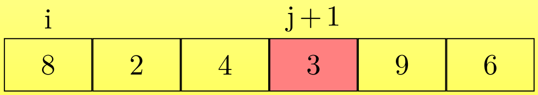
Example of Bubble Sort



Example of Bubble Sort



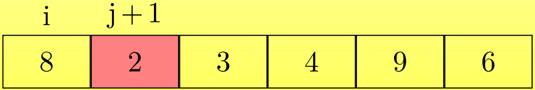
Example of Bubble Sort



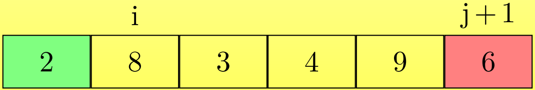
Example of Bubble Sort

i		j+1			
8	2	3	4	9	6

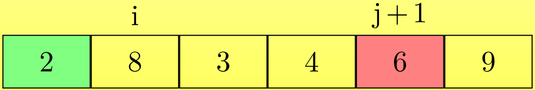
Example of Bubble Sort



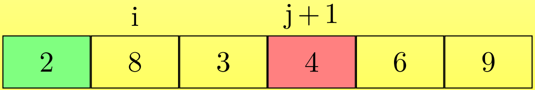
Example of Bubble Sort



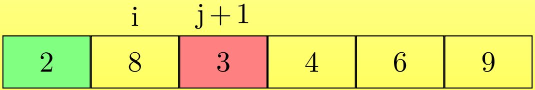
Example of Bubble Sort



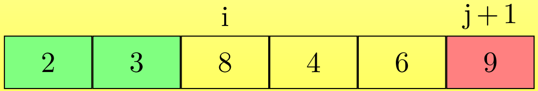
Example of Bubble Sort



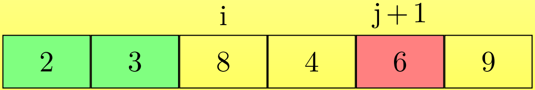
Example of Bubble Sort



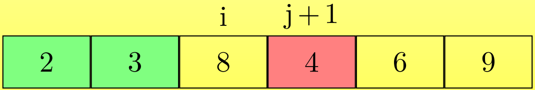
Example of Bubble Sort



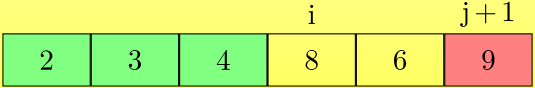
Example of Bubble Sort



Example of Bubble Sort



Example of Bubble Sort



Example of Bubble Sort

			i	j+1	
2	3	4	8	6	9

Example of Bubble Sort

2	3	4	6	8	9

Bubble Sort

- Time Complexity
  - Best: $O(n)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- Memory: 1
- Stable: Yes

Bubble Sort

- Time Complexity
  - Best: $O(n)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$
- Memory: 1
- Stable: Yes

Bubble-Sort(A)

```
for i = 1 to A.length − 1 do
    noswap = TRUE
    for j = A.length − 1 downto i do
        if A[j + 1] < A[j] then
            A[j] ↔ A[j + 1]
            noswap = FALSE
    if noswap then break
```

Contents

Shellsort

General idea:

- Choose a descending gap sequence (e.g., $D = [5, 3, 2, 1]$).
- In each round, elements with the same gap d are in the same group.
- Apply Insertion-Sort for each group.
- Reduce the amount of data migration that caused by insertion sort.

Shellsort

Shell-Pass(A, d)

```
for i = d + 1 to n do
    if A[i] < A[i − d] then
        key = A[i]        //A[i] is to inserted in the correct position
        j = i − d
        while j > 0 and key < A[j] do
            A[j + d] = A[j]
            j = j − d
        A[j + d] = key
```

Shellsort(A, D)

```
for increment in D do
    Shell-Pass(A, increment)
```

Example of Shellsort

21 25 49 <u>25</u> 16 08 27 04 55 48

Example of Shellsort

21 25 49 <u>25</u> 16 08 27 04 55 48 $d = 3$

Example of Shellsort

21	25	49	<u>25</u>	16	08	27	04	55	48	d = 3
21	04	08	<u>25</u>	16	49	27	25	55	48	

Example of Shellsort

21	25	49	<u>25</u>	16	08	27	04	55	48	$d = 3$
21	04	08	<u>25</u>	16	49	27	25	55	48	$d = 2$

Example of Shellsort

21	25	49	<u>25</u>	16	08	27	04	55	48	$d = 3$
21	04	08	<u>25</u>	16	49	27	25	55	48	$d = 2$
08	04	16	<u>25</u>	21	25	27	48	55	49	

Example of Shellsort

21	25	49	<u>25</u>	16	08	27	04	55	48	d = 3
21	04	08	<u>25</u>	16	49	27	25	55	48	d = 2
08	04	16	<u>25</u>	21	25	27	48	55	49	d = 1

Example of Shellsort

21	25	49	<u>25</u>	16	08	27	04	55	48	d = 3
21	04	08	<u>25</u>	16	49	27	25	55	48	d = 2
08	04	16	<u>25</u>	21	25	27	48	55	49	d = 1
04	08	16	21	<u>25</u>	25	27	48	49	55	

Shellsort

- Time Complexity
  - Best: depends on the gap sequence
  - Average: depends on the gap sequence
  - Worst: depends on the gap sequence, e.g., $O(n^{4/3})$, when the gap sequence is $4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1.
- Memory: 1
- Stable: No

Shellsort

Shellsort

Why Shellsort typically performs faster?

- Insertion-Sorting small-sized array although costs $O(n^2)$ in the worst case, but it is similar to $O(n)$ in values.
- For large array, when we use a gap large enough (in the order of $O(n)$), each sub-array has a small size, thus efficient to sort.
- After enough iterations, when the gap is small, the majority part of the array is already sorted (thus the complexity is small again).

Shellsort

How to select the gap sequence?

- $\lceil \frac{n}{2^k} \rceil$: time complexity $\Theta(n^2)$
- $2\lceil \frac{n}{2^{k+1}} \rceil + 1$: time complexity $\Theta(n^{\frac{3}{2}})$
- $2^k - 1$: time complexity $\Theta(n^{\frac{3}{2}})$
- $2^k + 1$ $(k \geq 0)$: time complexity $\Theta(n^{\frac{3}{2}})$
- Successive numbers of the form $2^p 3^q$ for prime numbers p, q: time complexity $\Theta(n \log^2 n)$.

Shellsort: the lowerbound on the time-complexity

The worst-case complexity of any version of Shellsort is of higher order: Plaxton, Poonen, and Suel showed that it grows at least as rapidly as $\boldsymbol{\Omega}\left(\mathrm{n}\left(\frac{\log \mathrm{n}}{\log \log \mathrm{n}}\right)^2\right)$.

Contents

Basic Concepts of Heap

Heap

A data structure which is an array object that can be viewed as a nearly complete binary tree.

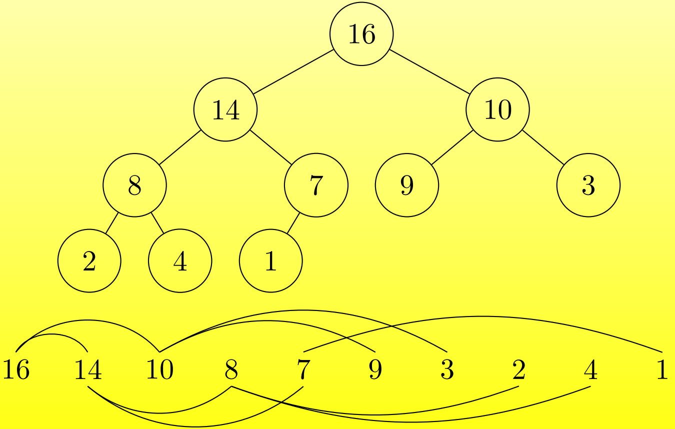
The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

Given the index i of a node, the indices of its parent Parent(i), left child Left(i), and right child Right(i) can be computed simply:

Parent(i)	return	$\lfloor i/2 \rfloor$
Left(i)	return	$2*i$
Right(i)	return	$2*i+1$

Example of Max-heap

max-heap: A[Parent(i)] $\geq$ A[i], for all i other than the root.



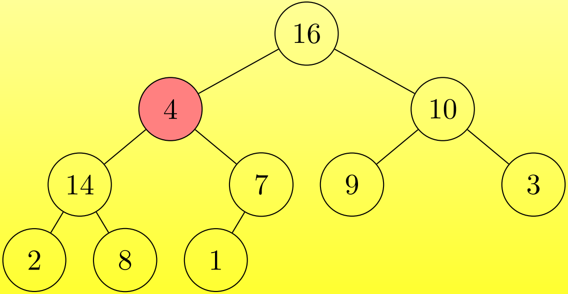
Maintaining the Heap Property

Assumption: sub-trees rooted at Left(i) & Right(i) are max-heaps.

```
Max-Heapify(A, i)  // Input an an array and an index i
l = Left(i);
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i] then
      largest = l
else      largest = i
if r ≤ A.heap-size and A[r] > A[largest] then
      largest = r
if largest ≠ i then
      A[i] ↔ A[largest]
      Max-Heapify(A, largest)
```

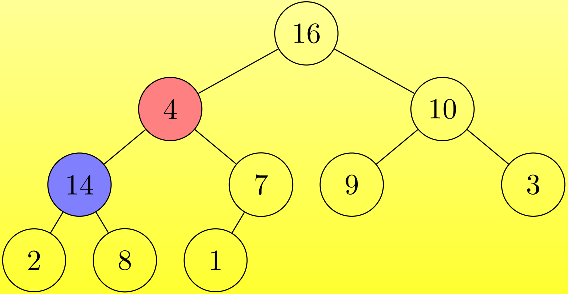
Maintaining the Heap Property

Example: MAX-HEAPIFY(A, 2)



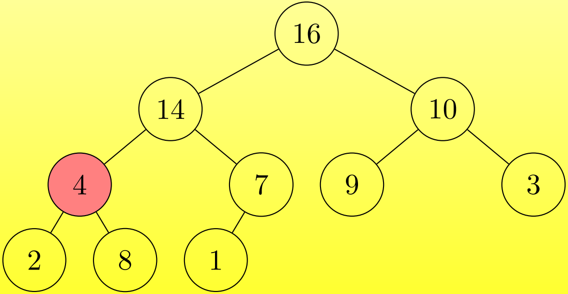
Maintaining the Heap Property

Example: MAX-HEAPIFY(A, 2)



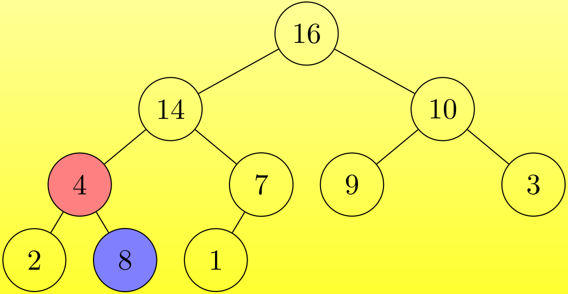
Maintaining the Heap Property

Example: MAX-HEAPIFY(A, 2)



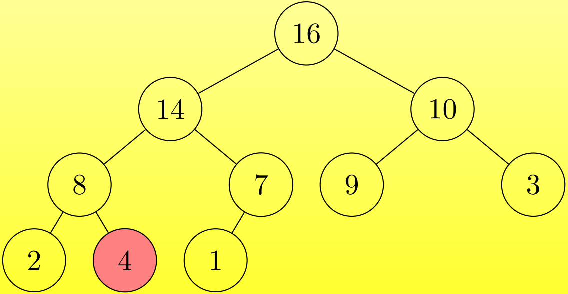
Maintaining the Heap Property

Example: MAX-HEAPIFY(A, 2)



Maintaining the Heap Property

Example: MAX-HEAPIFY(A, 2)



Maintaining the Heap Property

Assumption: sub-trees rooted at Left(i) & Right(i) are max-heaps.

```
Max-Heapify(A, i)  // Input an an array and an index i
l = Left(i);
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i] then
    largest = l
else    largest = i
if r ≤ A.heap-size and A[r] > A[largest] then
    largest = r
if largest ≠ i then
    A[i] ↔ A[largest]
    Max-Heapify(A, largest)
```

Building a Heap

Fact: with the array representation of an n-element heap, the leaves are the nodes indexed from $\lfloor A.length/2 \rfloor + 1$ to n, and each leaf is a 1-element max-heap to begin with.

Building a Heap

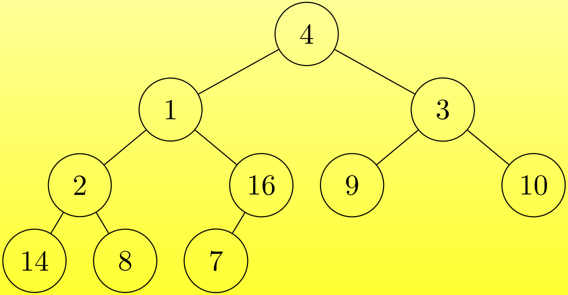
Fact: with the array representation of an n-element heap, the leaves are the nodes indexed from $\lfloor A.length/2 \rfloor + 1$ to n, and each leaf is a 1-element max-heap to begin with.

```
Build-Max-Heap(A)
1: A.heap-size = A.length
2: for i = ⌊A.length/2⌋ downto 1 do
3:     Max-Heapify(A, i)
```

Building a Heap

$A.\text{length} = 10$

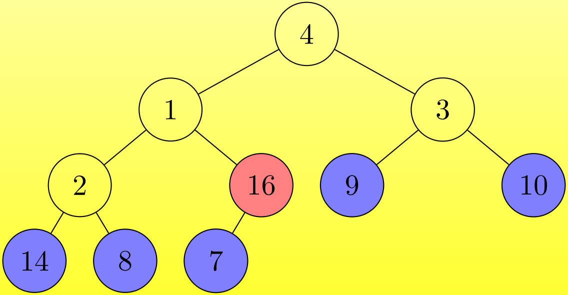
4	1	3	2	16	9	10	14	8	7



Building a Heap

A.length = 10

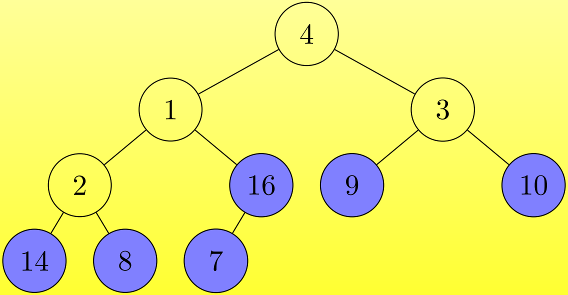
4	1	3	2	16	9	10	14	8	7



Building a Heap

A.length = 10

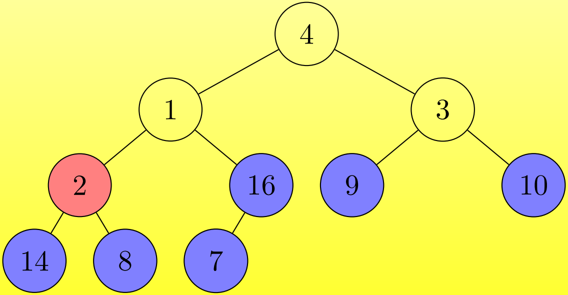
4	1	3	2	16	9	10	14	8	7



Building a Heap

A.length = 10

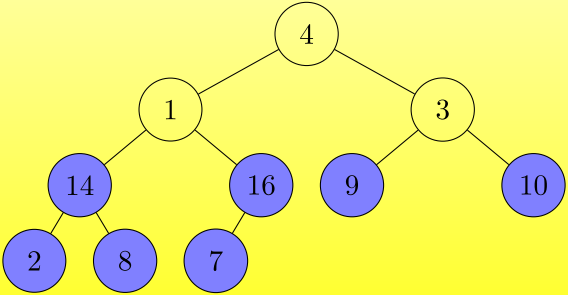
4	1	3	2	16	9	10	14	8	7



Building a Heap

A.length = 10

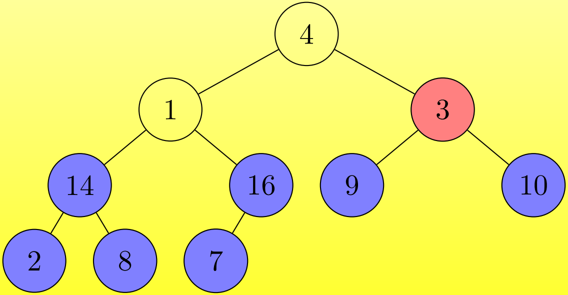
4	1	3	14	16	9	10	2	8	7



Building a Heap

A.length = 10

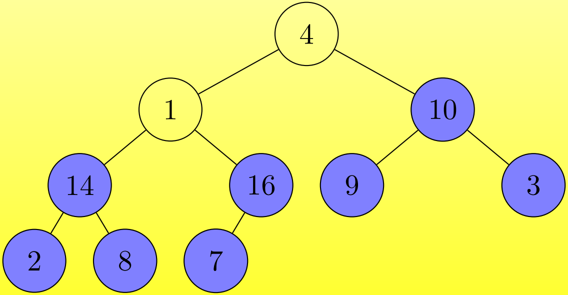
4 1 3 14 16 9 10 2 8 7



Building a Heap

A.length = 10

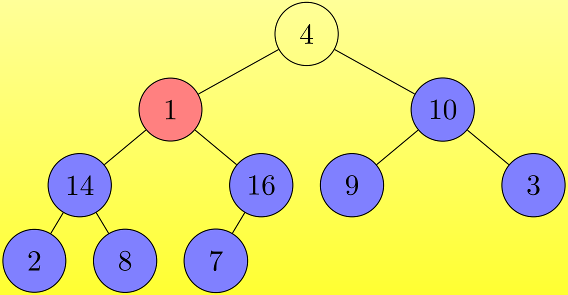
4	1	10	14	16	9	3	2	8	7



Building a Heap

A.length = 10

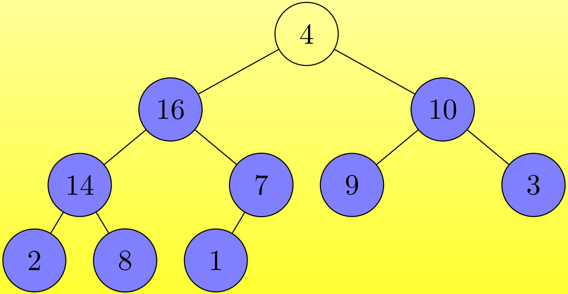
4 1 10 14 16 9 3 2 8 7



Building a Heap

A.length = 10

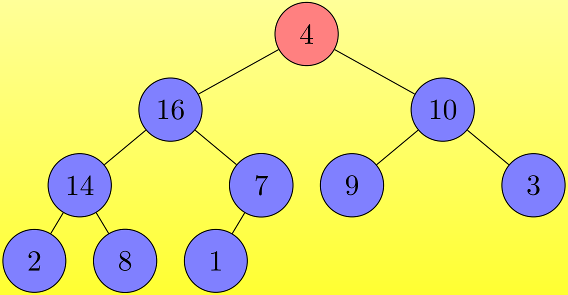
4 16 10 14 7 9 3 2 8 1



Building a Heap

A.length = 10

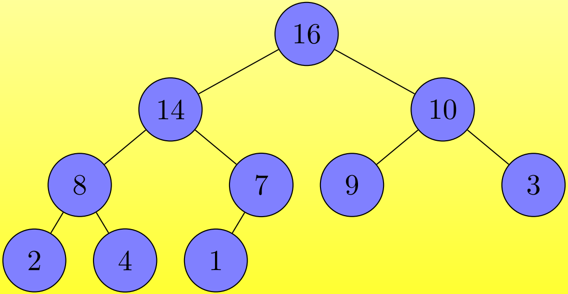
4	16	10	14	7	9	3	2	8	1



Building a Heap

A.length = 10

16 14 10 8 7 9 3 2 4 1



Building a Heap

Fact: with the array representation of an n-element heap, the leaves are the nodes indexed from $\lfloor A.length/2 \rfloor + 1$ to n, and each leaf is a 1-element max-heap to begin with.

Building a Heap

Fact: with the array representation of an n-element heap, the leaves are the nodes indexed from $\lfloor A.length/2 \rfloor + 1$ to n, and each leaf is a 1-element max-heap to begin with.

Build-Max-Heap(A)

```
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1 do
    Max-Heapify(A, i)
```

The Heapsort Algorithm

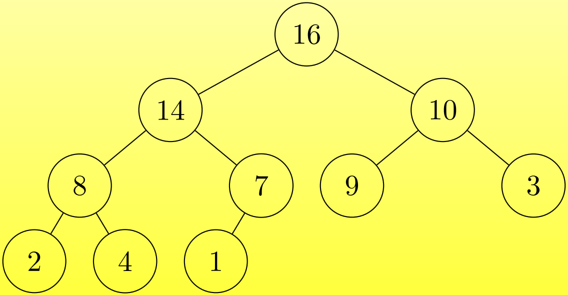
General idea: Same as selection sort, maintain the minimum (maximum) element by using heap.
MAX-HEAP: A[1] always stores the largest number.

The Heapsort Algorithm

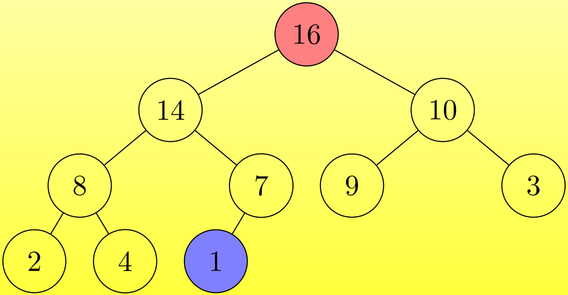
General idea: Same as selection sort, maintain the minimum (maximum) element by using heap.
MAX-HEAP: A[1] always stores the largest number.

```
Heapsort(A)
Build-Max-Heap(A)
for i = A.length downto 2 do
    A[1] ↔ A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

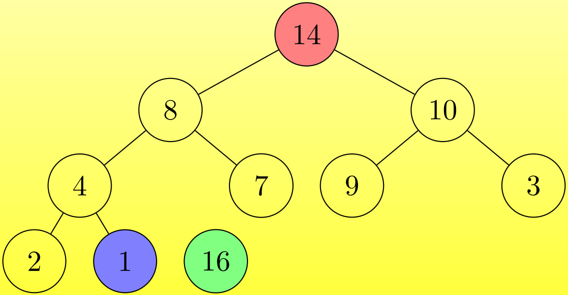
Example of Heapsort



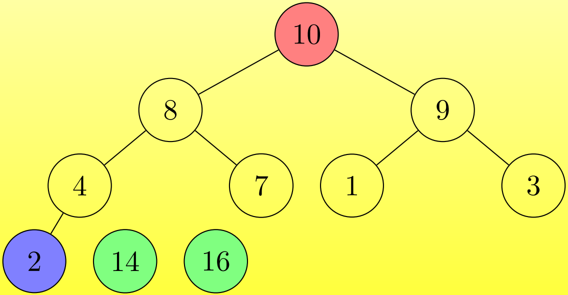
Example of Heapsort



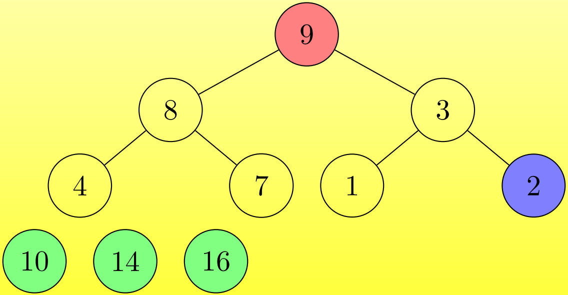
Example of Heapsort



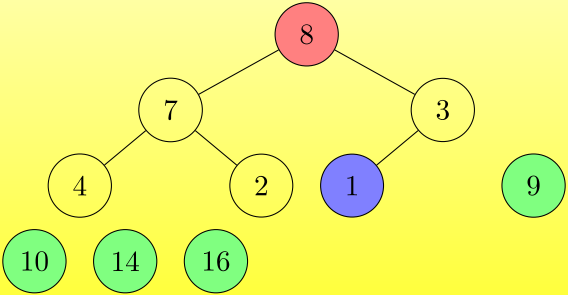
Example of Heapsort



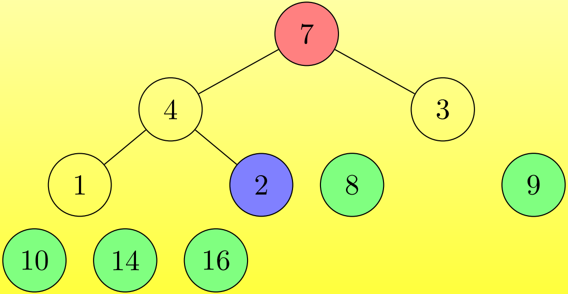
Example of Heapsort



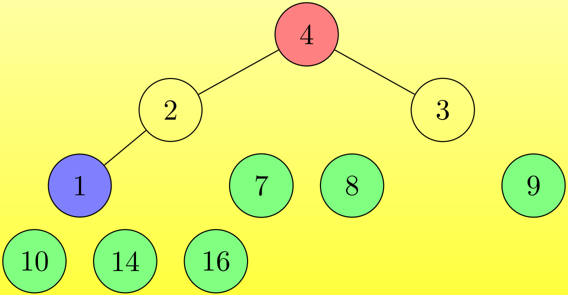
Example of Heapsort



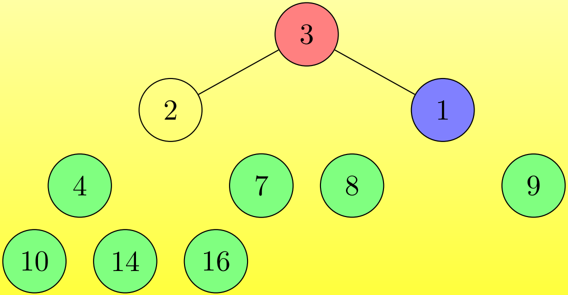
Example of Heapsort



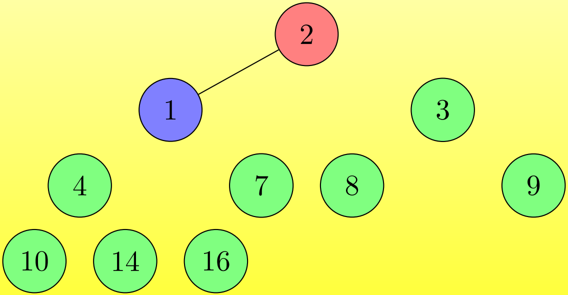
Example of Heapsort



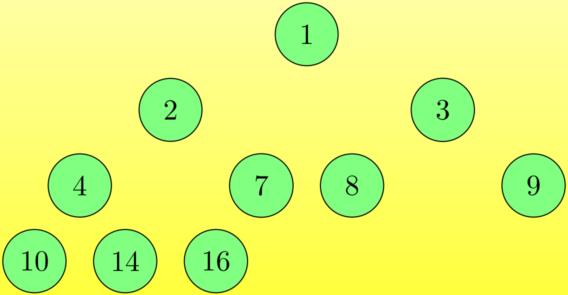
Example of Heapsort



Example of Heapsort



Example of Heapsort



Heapsort

- Time Complexity
  - Max-Heapify: $O(\log n)$ – Why?
  - Build-Max-Heap: $O(n)$ – Why?
  - Best: $O(n \log n)$
  - Average: $O(n \log n)$
  - Worst: $O(n \log n)$
- Memory: 1
- Stable: No

Priority Queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:

- Insert(S, x) inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.
- Maximum(S) returns the element of S with the largest key.
- Extract-Max(S) removes and returns the element of S with the largest key.
- Increase-Key(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Priority Queues

Heap-Maximum(A)

```
return A[1]
```

Heap-Extract-Max(A)

```
if A.heap-size < 1 then
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
Max-Heapify(A, 1)
return max
```

Priority Queues

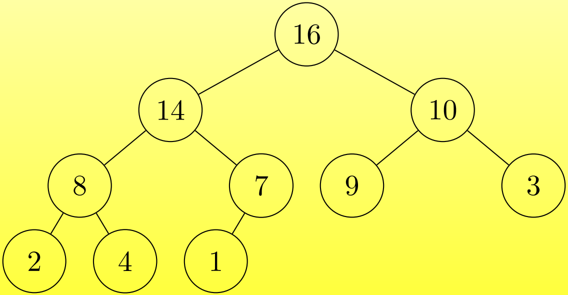
```
Heap-Increase-Key(A, i, key)
if key < A[i] then
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[Parent(i)] < A[i] do
    A[i] ↔ A[Parent(i)]
    i = Parent(i)
```

Priority Queues

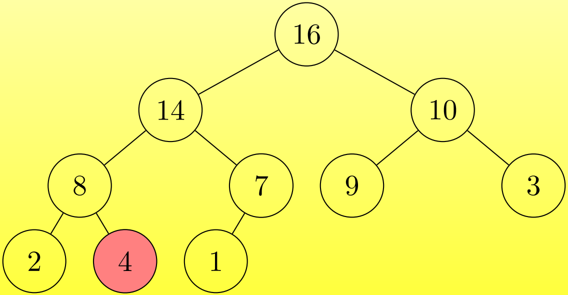
Max-Heap-Insert(A, key)

1: A.heap-size = A.heap-size + 1
2: A[A.heap-size] = $-\infty$
3: Heap-Increase-Key(A, A.heap-size, key)

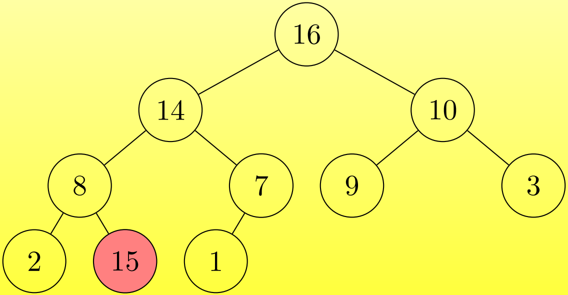
Example of Heap-Increase-Key



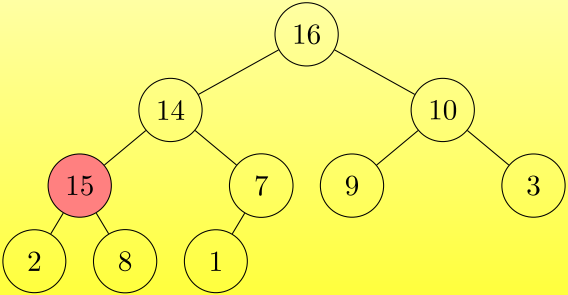
Example of Heap-Increase-Key



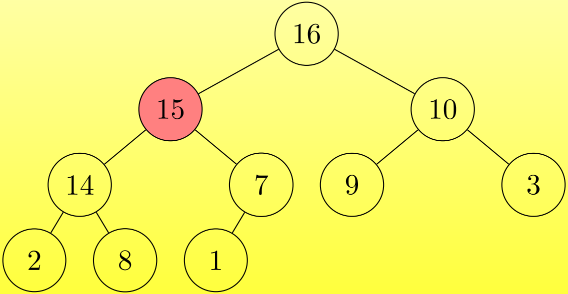
Example of Heap-Increase-Key



Example of Heap-Increase-Key



Example of Heap-Increase-Key



Priority Queues

Heap-Increase-Key(A, i, key)

```
if key < A[i] then
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[Parent(i)] < A[i] do
    A[i] ↔ A[Parent(i)]
    i = Parent(i)
```

Contents

Quicksort

General idea:

- Arbitrarily choose an element x in the unsorted set for comparison.
- Divide the unsorted elements into two parts: $\leq$ x and $>$ x.
- Recursively use Quicksort for the above two parts.

Quicksort

General idea:

- Arbitrarily choose an element x in the unsorted set for comparison.
- Divide the unsorted elements into two parts: $\leq x$ and $> x$.
- Recursively use Quicksort for the above two parts.

```
Quicksort(A, p, r)
if p < r then
    q = Partition(A, p, r)
    Quicksort(A, p, q − 1)
    Quicksort(A, q + 1, r)
```

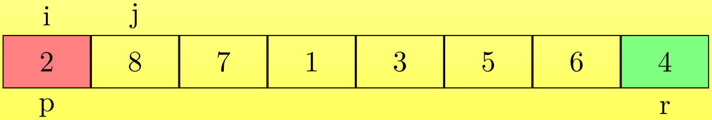
Partition

```
Partition(A, p, r)
x = A[r]        // pivot element
i = p − 1
for j = p to r − 1 do
    if A[j] ≤ x then
        i = i + 1
        A[i] ↔ A[j]
A[i + 1] ↔ A[r]
return i + 1
```

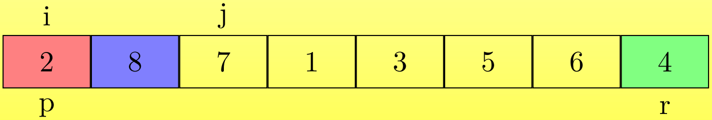
Example of Partition

i	j							
	2	8	7	1	3	5	6	4
	p							r

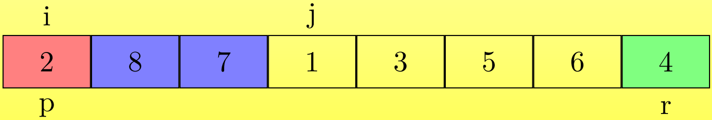
Example of Partition



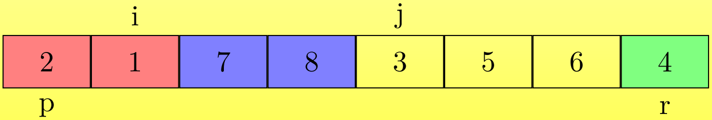
Example of Partition



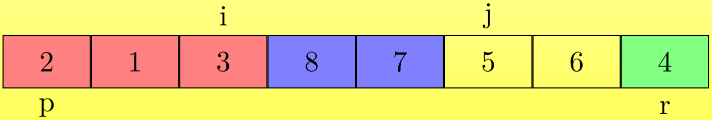
Example of Partition



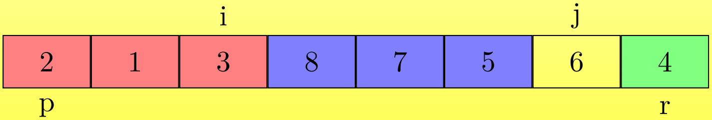
Example of Partition



Example of Partition



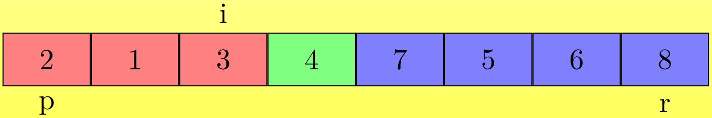
Example of Partition



Example of Partition

		i					
2	1	3	8	7	5	6	4
p							r

Example of Partition



Performance of Quicksort

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements. The partitioning costs $\Theta(n)$ time. the recurrence for the running time is

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2).
\end{aligned}
$$

Performance of Quicksort

Best-case partitioning

In the most even possible split, Partition produces two subproblems, each of size no more than n/2, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. In this case, quicksort runs much faster. The recurrence for the running time is then

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

Performance of Quicksort

Balanced partitioning

What if the split is always $\frac{1}{10} : \frac{9}{10}$? The recurrence for the running time is

$$\begin{aligned} T(n) &= T(\frac{1}{10}n) + T(\frac{9}{10}n) + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

A Randomized Version of Quicksort

```
Randomized-Partition(A, p, r)
i = Random(p, r)
A[r] ↔ A[i]
return Partition(A, p, r)
```

```
Randomized-Quicksort(A, p, r)
if p < r then
    q = Randomized-Partition(A, p, r)
    Randomized-Quicksort(A, p, q − 1)
    Randomized-Quicksort(A, q + 1, r)
```

Analysis of Quicksort

Worst-case analysis

We saw that a worst-case split at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which, intuitively, is the worst-case running time of the algorithm.

Using the substitution method (see Section 4.3), we can show that the running time of quicksort is $O(n^2)$.

Analysis of Quicksort

Let $T(n)$ be the worst-case time for the procedure Quicksort on an input of size n. We have

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) \\ &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2 + \Theta(n)) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2 + \Theta(n)) \\ &\le cn^2 - c(2n-1) + \Theta(n) \le cn^2. \end{aligned}$$

Analysis of Quicksort

Running time and comparisons

Rename the elements of the array A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element (assuming distinct elements). $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ to be the set of elements between z_i and z_j.

We define

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

Since each pair is compared at most once, we can easily characterize the total number of comparisons performed by the algorithm:

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

Analysis of Quicksort

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

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

Analysis of Quicksort

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n).
\end{aligned}
$$

Quicksort

- Time Complexity
  - Best: $O(n \log n)$
  - Average: $O(n \log n)$
  - Worst: $O(n^2)$
- Memory: $O(\log n)$ on average, worst case space complexity is $O(n)$
- Stable: stable versions exist

Summary

Name	Average	Worst	Stable	Method
Insertion Sort	$O(n^2)$	$O(n^2)$	Yes	Insertion
Selection Sort	$O(n^2)$	$O(n^2)$	No	Selection
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Exchanging
Merge sort	$O(n \log n)$	$O(n \log n)$	Yes	Merging
Shellsort	(*)	$O(n^{4/3})$ (*)	No	Insertion
Heapsort	$O(n \log n)$	$O(n \log n)$	No	Selection
Quicksort	$O(n \log n)$	$O(n^2)$	Exist	Partitioning

*The time complexity of shellsort depends on the selected gap sequence.

A sorting algorithm animation website:

https://www.toptal.com/developers/sorting-algorithms