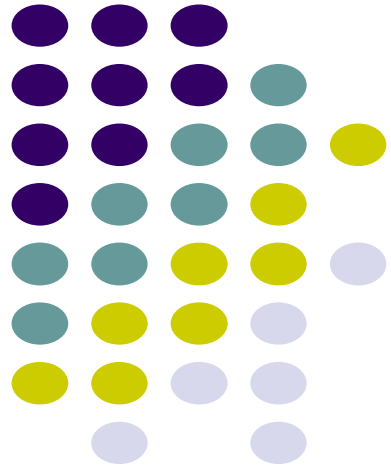
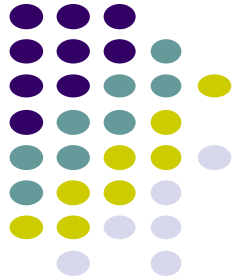
Comparison based Sorting Algorithms

Dr. Navjot Singh

Design and Analysis of Algorithms



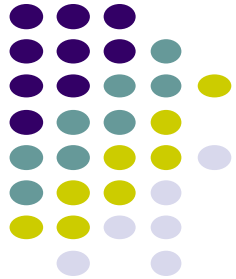
Sorting – Definitions



- Input: n *records*, $R_1 \ldots R_n$, from a *file*.
- Each record R_i has
  - a key K_i
  - possibly other (satellite) information
- The keys must have an ordering relation $\prec$ that satisfies the following properties:
  - *Trichotomy*: For any two keys a and b, exactly one of $a \prec b$, $a = b$, or $a \succ b$ is true.
  - *Transitivity*: For any three keys a, b, and c, if $a \prec b$ and $b \prec c$, then $a \prec c$.

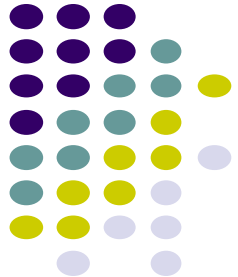
  The relation $\preceq$ is a *total ordering* (*linear ordering*) on keys.

Sorting – Definitions



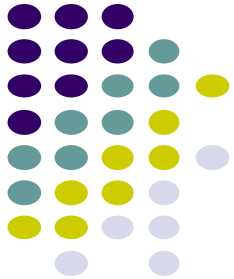
- *Sorting*: determine a permutation $\Pi = (p_1, \ldots, p_n)$ of n records that puts the keys in non-decreasing order $K_{p_1} \leq \ldots \leq K_{p_n}$.
- *Permutation*: a one-to-one function from {1, …, n} onto itself. There are $n!$ distinct permutations of n items.
- *Rank*: Given a collection of n keys, the *rank* of a key is the number of keys that precede it. That is, $rank(K_j) = |\{K_i | K_i < K_j\}|$. If the keys are distinct, then the rank of a key gives its position in the output file.

Sorting Terminology



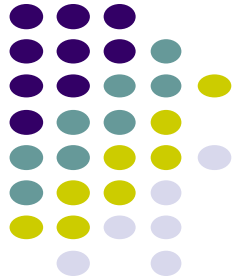
- *Internal* (the file is stored in main memory and can be randomly accessed) vs. *External* (the file is stored in secondary memory & can be accessed sequentially only)
- *Comparison-based sort*: uses only the relation among keys, not any special property of the representation of the keys themselves.
- *Stable sort*: records with equal keys retain their original relative order; i.e., $i < j$ & $Kp_i = Kp_j \Rightarrow p_i < p_j$
- *Array-based* (consecutive keys are stored in consecutive memory locations) *vs. List-based sort* (may be stored in nonconsecutive locations in a linked manner)
- *In-place sort*: needs only a constant amount of extra space in addition to that needed to store keys.

Sorting Categories



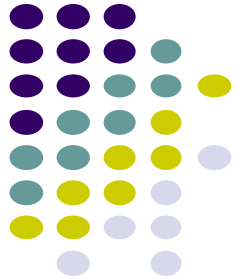
- Sorting by Insertion — *insertion sort, shellsort*
- Sorting by Exchange — *bubble sort, quicksort*
- Sorting by Selection — *selection sort, heapsort*
- Sorting by Merging — *merge sort*
- Sorting by Distribution — *radix sort*

Elementary Sorting Methods



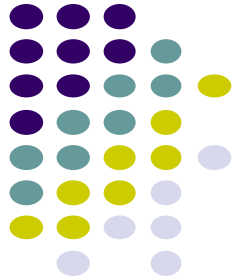
- Easier to understand the basic mechanisms of sorting.
- Good for small files.
- Good for well-structured files that are relatively easy to sort, such as those almost sorted.
- Can be used to improve efficiency of more powerful methods.

Selection Sort



```
Selection-Sort(A, n)
1.  for i = 1 to n do
2.        min ← i
3.        for j = i + 1 to n do
4.              if A[min] > A[j] then
5.                    min ← j
6.         t ← A[min]
7.        A[min] ← A[i]
8.        A[i] ← t
```

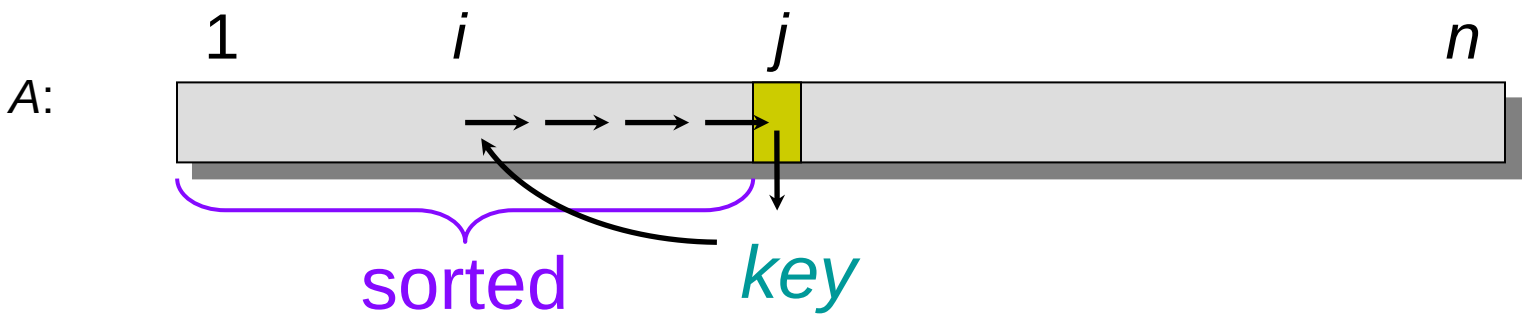
Algorithm Analysis



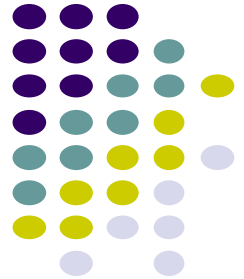
- Is it *in-place*?
- Is it *stable*?
- The number of comparisons is $\Theta(n^2)$ in all cases.
- Can be improved by a some modifications, which leads to heapsort.

Insertion Sort

```
InsertionSort(A, n)
1.  for j = 2 to n do
2.      key ← A[j]
3.      i ← j – 1
4.      while i > 0 and key < A[i]
5.          A[i+1] ← A[i]
6.          i ← i – 1
7.      A[i+1] ← key
```



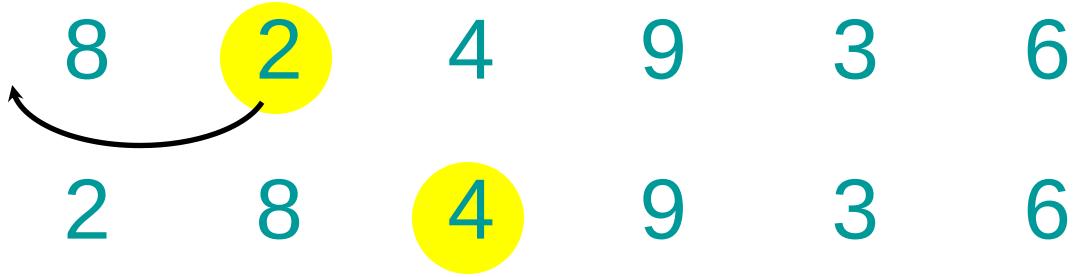
Example of insertion sort



8 2 4 9 3 6

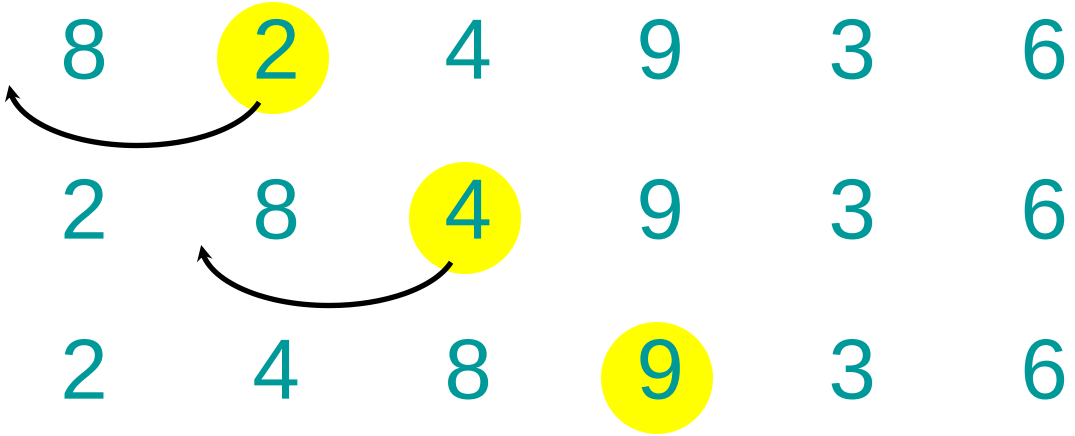
Example of insertion sort

Example of insertion sort

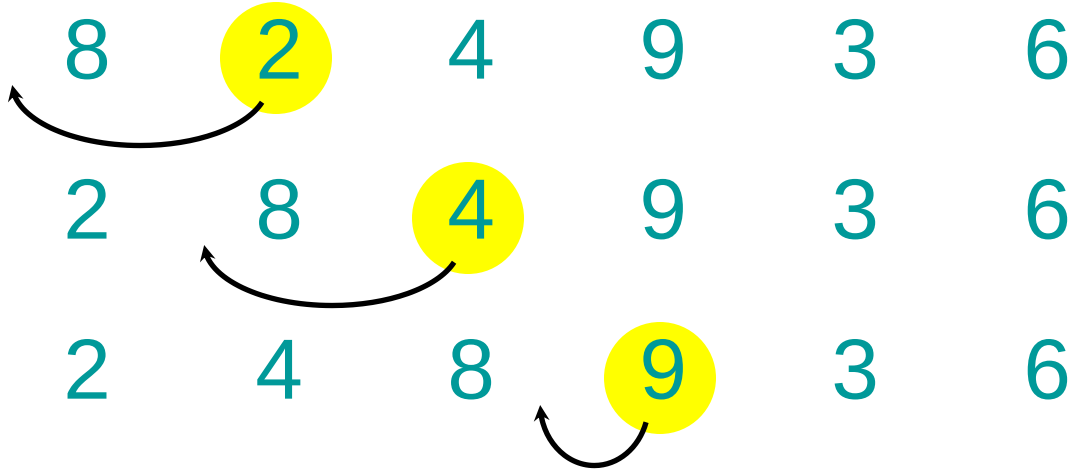


Example of insertion sort

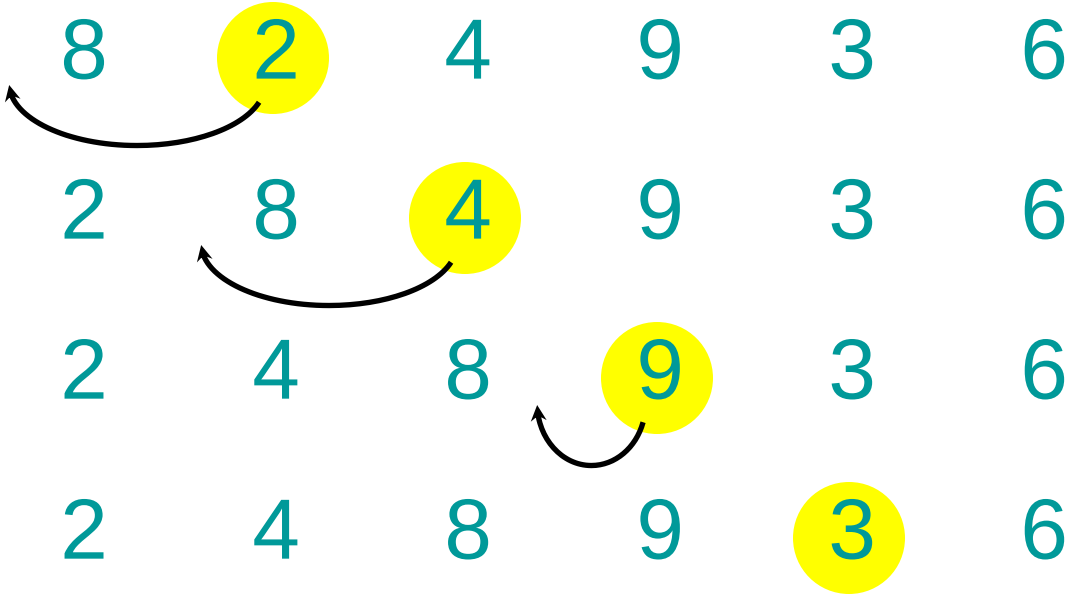
Example of insertion sort



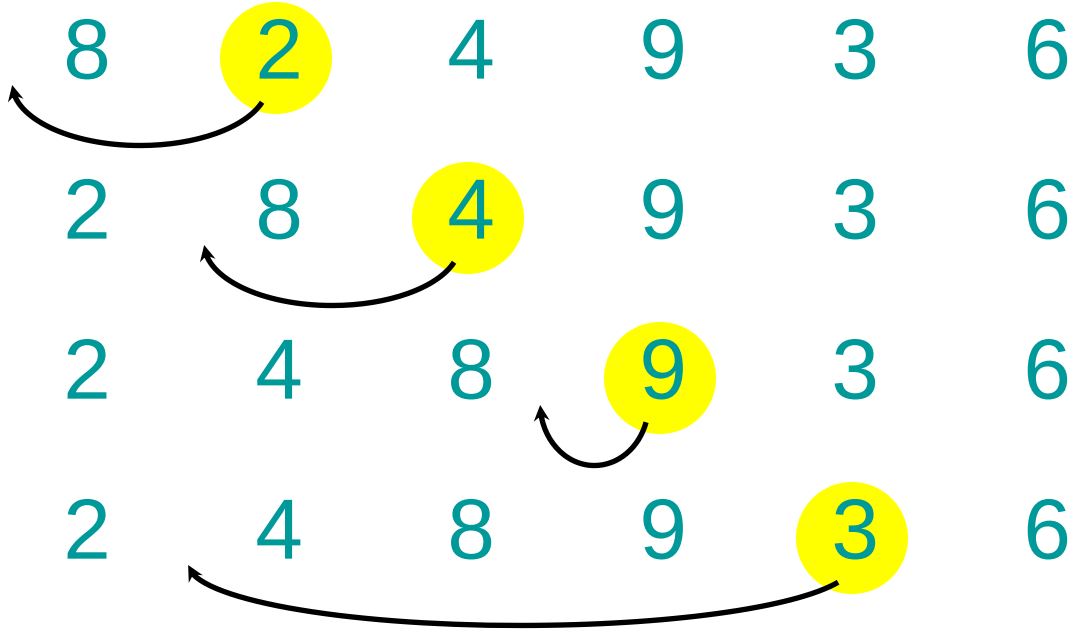
Example of insertion sort



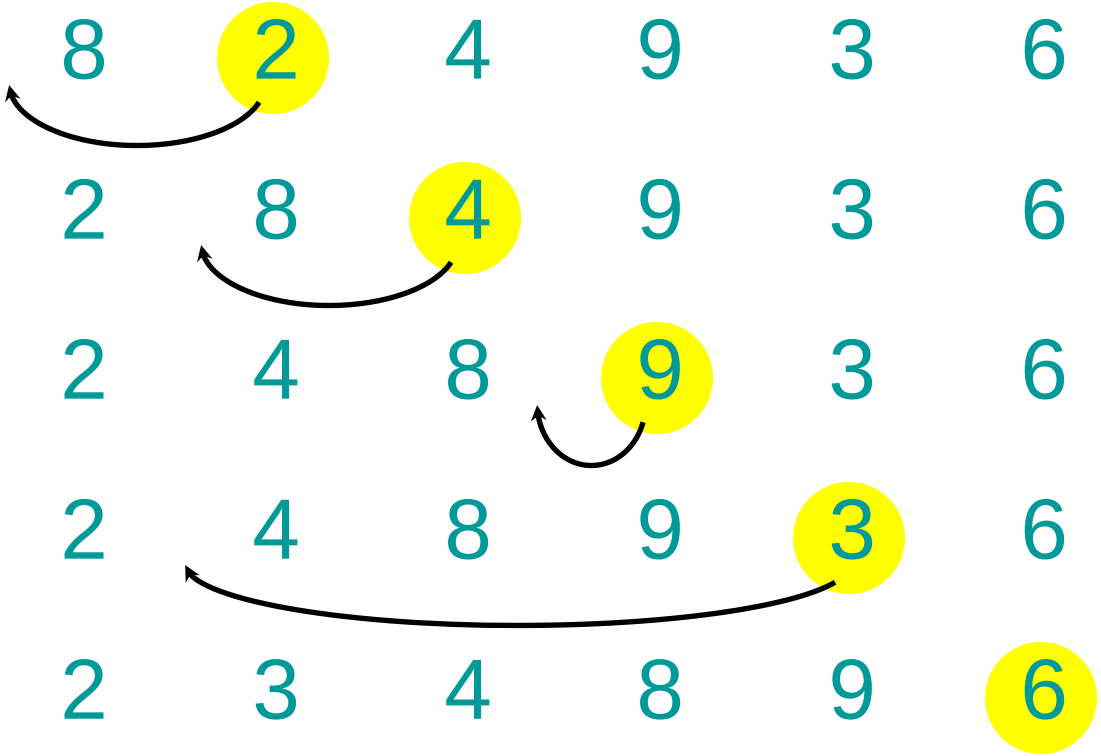
Example of insertion sort



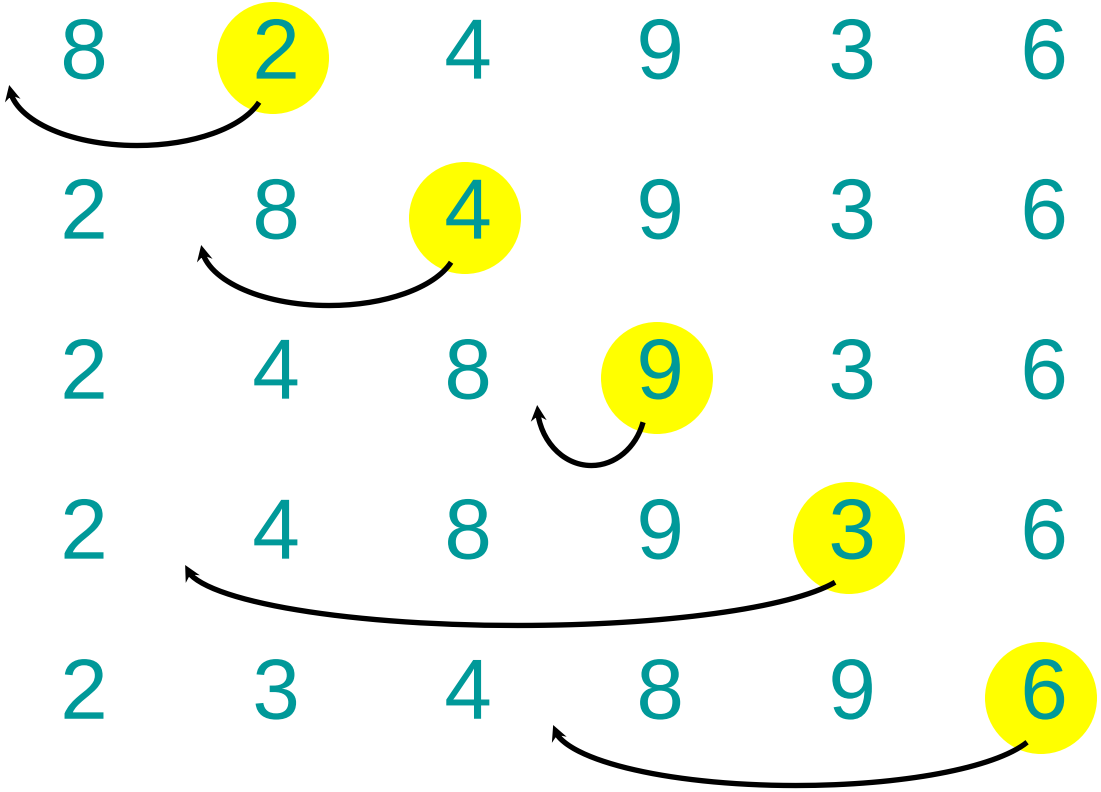
Example of insertion sort



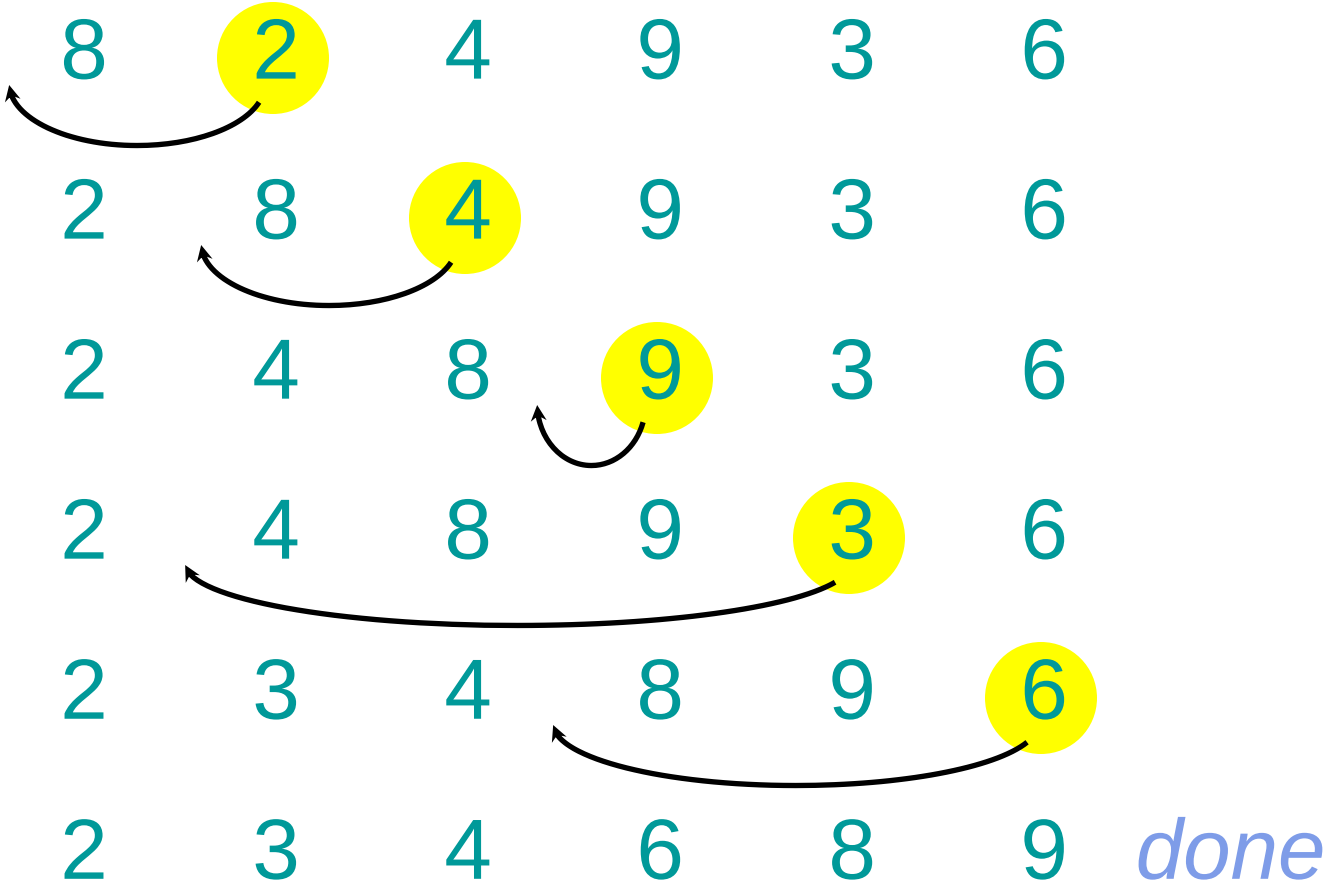
Example of insertion sort



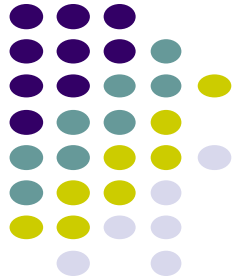
Example of insertion sort



Example of insertion sort

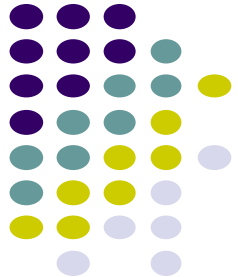


Algorithm Analysis



- Is it *in-place*?
- Is it *stable*?
- No. of Comparisons:
  - If A is sorted: $\Theta(n)$ comparisons
  - If A is reverse sorted: $\Theta(n^2)$ comparisons
  - If A is randomly permuted: $\Theta(n^2)$ comparisons

Worst-case Analysis

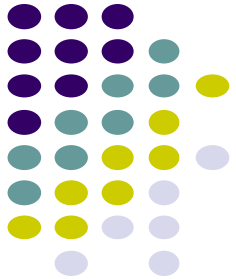


- The *maximum* number of comparisons while inserting $A[i]$ is $(i\text{-}1)$. So, the number of comparisons is

$$
\begin{aligned}
C_{wc}(n) &\le \sum_{i = 2 \text{ to } n} (i - 1) \\
&= \sum_{j = 1 \text{ to } n-1} j \\
&= n(n-1)/2 \\
&= \Theta(n^2)
\end{aligned}
$$

- For which input does insertion sort perform $n(n-1)/2$ comparisons?

Average-case Analysis



- Want to determine the average number of comparisons taken over all possible inputs.
- Determine the average no. of comparisons for a key $A[j]$.
- $A[j]$ can belong to any of the j locations, $1..j$, with equal probability.
- The number of *key* comparisons for $A[j]$ is $j–k+1$, if $A[j]$ belongs to location k, $1 < k \leq j$ and is $j–1$ if it belongs to location 1.

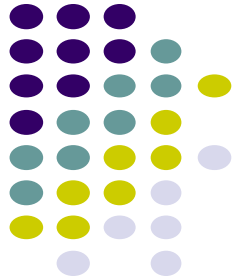
Average no. of comparisons for inserting key $A[j]$ is:

$$\sum_{k=1}^{j-1}\left(\frac{1}{j}k\right)+\frac{1}{j}(j-1)$$

$$=\frac{1}{j}\sum_{k=1}^{j-1}(k)+1-\frac{1}{j}$$

$$=\frac{j-1}{2}+1-\frac{1}{j}=\frac{j+1}{2}-\frac{1}{j}$$

Average-case Analysis



Summing over the no. of comparisons for all keys,

$$C_{avg}(n) = \sum_{i=2}^{n} \left(\frac{i+1}{2} - \frac{1}{i} \right)$$

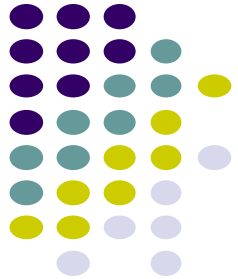
$$= \frac{n^2}{4} + \frac{3n}{4} - 1 - \sum_{i=2}^{n} \frac{1}{i}$$

$$= \Theta(n^2) - O(\ln n)$$

$$= \Theta(n^2)$$

Therefore, $T_{avg}(n) = \Theta(n^2)$

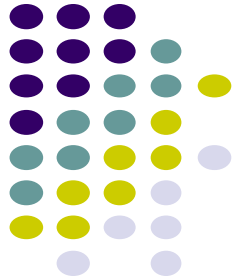
Analysis of Inversions in Permutations



- Inversion: A pair (i, j) is called an inversion of a permutation Π if $i < j$ and $\Pi(i) > \Pi(j)$.
- Worst Case: $n(n-1)/2$ inversions. For what permutation?
- Average Case:
  - Let $\Pi^T = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$ be the transpose of Π.
  - Consider the pair (i, j) with $i < j$, there are $n(n-1)/2$ pairs.
  - (i, j) is an inversion of Π if and only if $(n-j+1, n-i+1)$ is not an inversion of Π^T.
  - This implies that the pair (Π, Π^T) together have $n(n-1)/2$ inversions.

  $\Rightarrow$ The average number of inversions is $n(n-1)/4$.

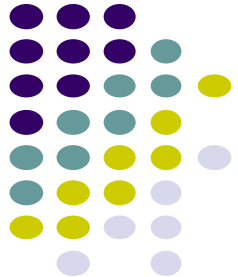
Theorem



Theorem: Any algorithm that sorts by comparison of keys and removes at most one inversion after each comparison must do at least $n(n-1)/2$ comparisons in the worst case and at least $n(n-1)/4$ comparisons on the average.

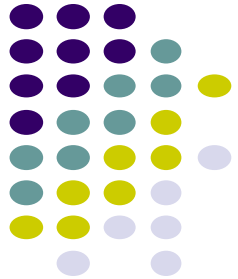
So, if we want to do better than $\Theta(n^2)$, we have to *remove more than a constant number of inversions* with each comparison.

Shellsort



- Simple *extension* of insertion sort.
- Gains speed by allowing exchanges with elements that are far apart (thereby fixing multiple inversions).
- *h-sort* the file
  - Divide the file into *h* subsequences.
  - Each subsequence consists of keys that are *h* locations apart in the input file.
  - Sort each *h*-sequence using insertion sort.
  - Will result in *h* *h*-sorted files. Taking every *h*th key from anywhere results in a sorted sequence.
- *h*-sort the file for decreasing values of increment *h*, with *h*=1 in the last iteration.
- *h*-sorting for large values of *h* in earlier iterations, *reduces* the number of comparisons for smaller values of *h* in later iterations.
- Correctness follows from the fact that the last step is plain insertion sort.

Shellsort



- A *family* of algorithms, characterized by the sequence $\{h_k\}$ of increments that are used in sorting.
- By interleaving, we can fix multiple inversions with each comparison, so that later passes see files that are “nearly sorted.” This implies that either there are many keys not too far from their final position, or only a small number of keys are far off.

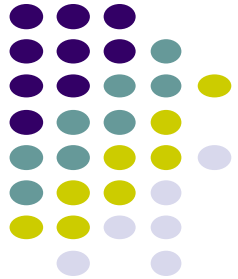
Shellsort

```
ShellSort(A, n)
1.  h ← 1
2.  while h ≤ n {
3.      h ← 3h + 1
4.   }
5.  repeat
6.      h ← h/3
7.      for i = h to n do {
8.          key ← A[i]
9.          j ← i
10.         while key < A[j - h] {
11.             A[j] ← A[j - h]
12.             j ← j - h
13.             if j < h then break
14.         }
15.         A[j] ← key
16.     }
17. until h ≤ 1
```

When h=1, this is insertion sort.
Otherwise, performs insertion sort on keys h locations apart.

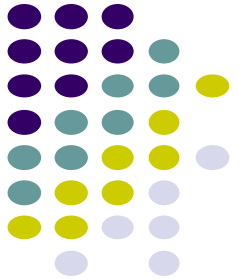
h values are set in the outermost repeat loop. Note that they are *decreasing* and the final value is 1.

Algorithm Analysis



- In-place sort
- Not stable
- The exact behavior of the algorithm depends on the sequence of increments -- difficult & complex to analyze the algorithm.
- For $h_k = 2^k - 1$, $T(n) = \Theta(n^{3/2})$

Quicksort

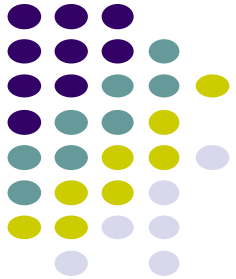


```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

```
HOARE-PARTITION(A, p, r)
 1  x ← A[p]
 2  i ← p − 1
 3  j ← r + 1
 4  while TRUE
 5        do repeat j ← j − 1
 6              until A[j] ≤ x
 7              repeat i ← i + 1
 8              until A[i] ≥ x
 9              if i < j
10                    then exchange A[i] ↔ A[j]
11                    else return j
```

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



What does it do?

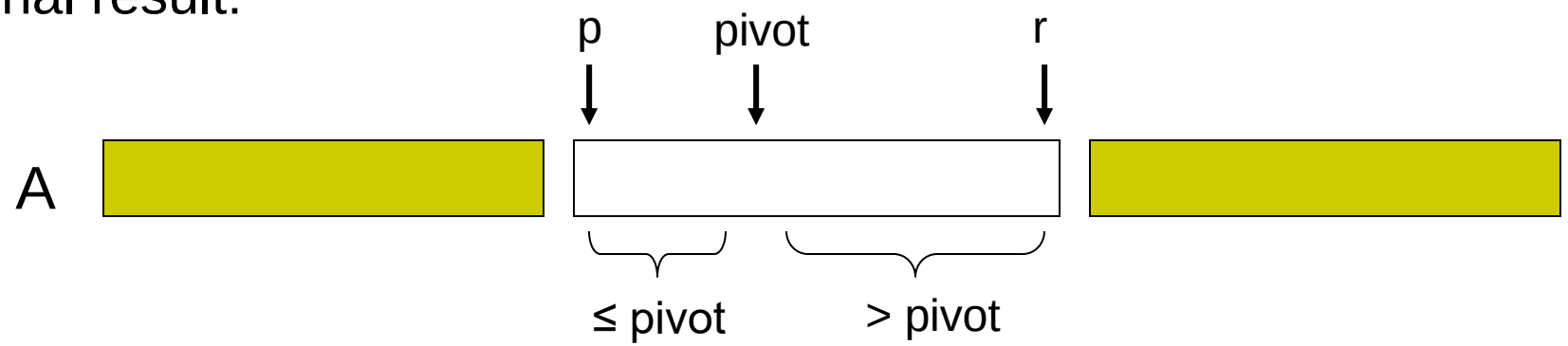
A[r] is called the ***pivot***

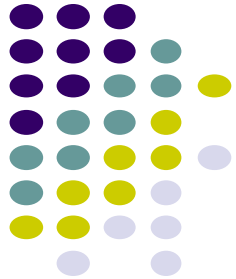
Partitions the elements A[p…r-1] in to two sets, those ≤ pivot and those > pivot

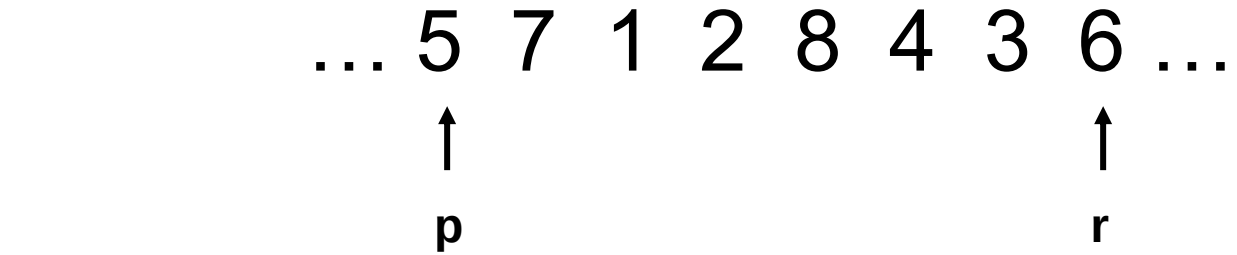
Operates in place

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

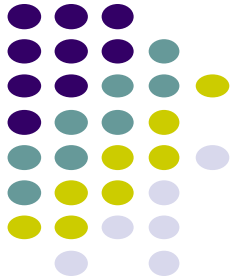
Final result:

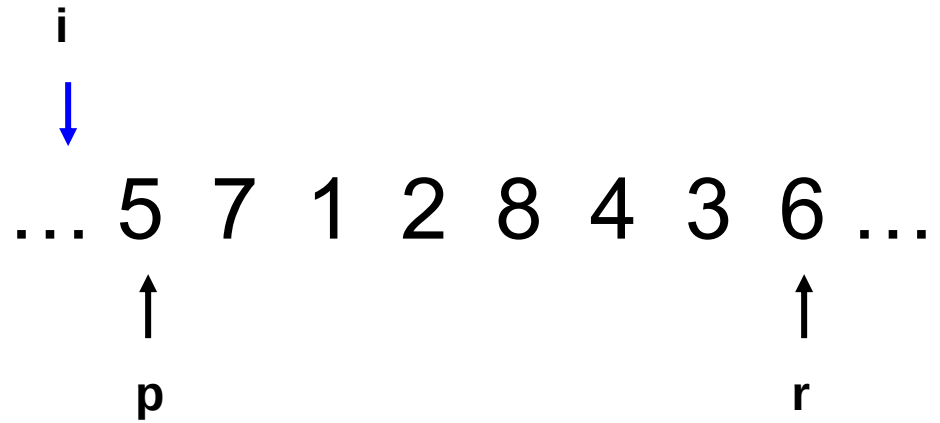




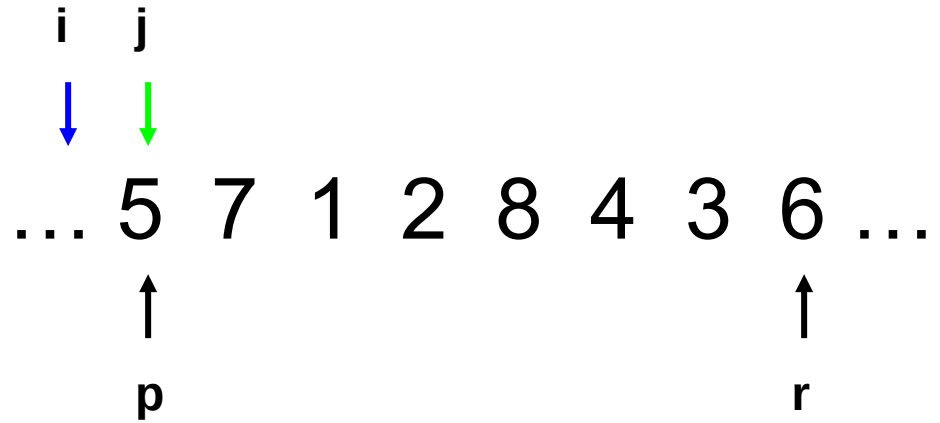



```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

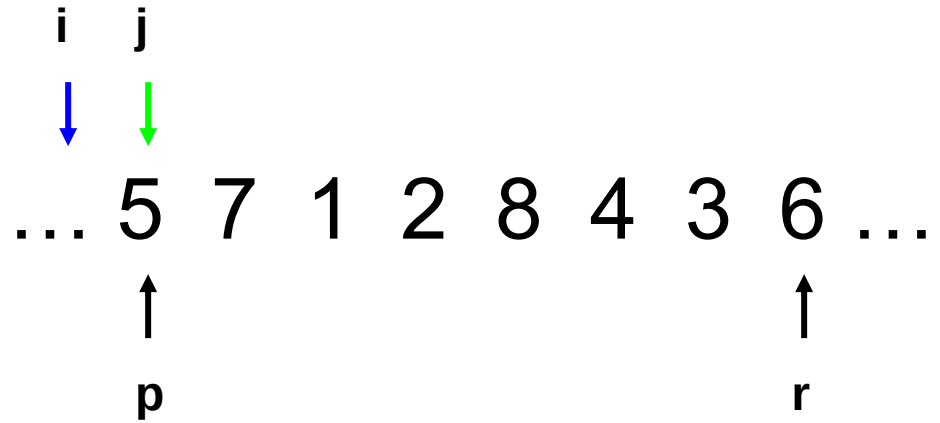




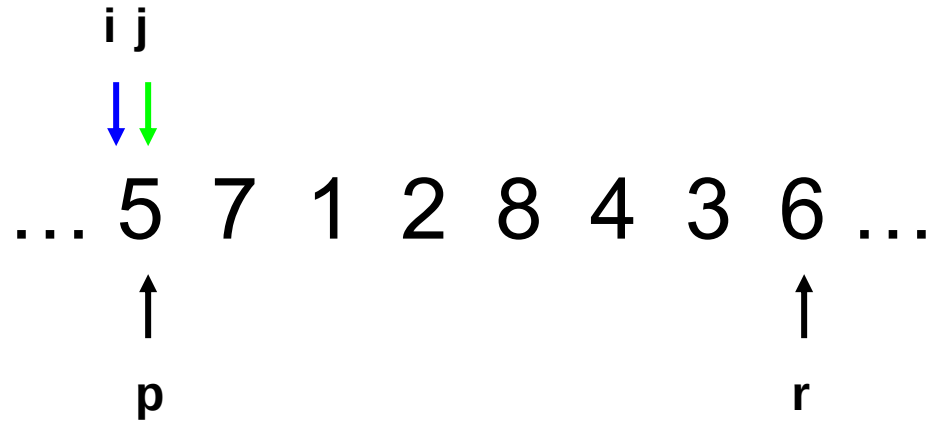

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



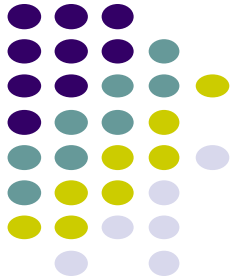

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

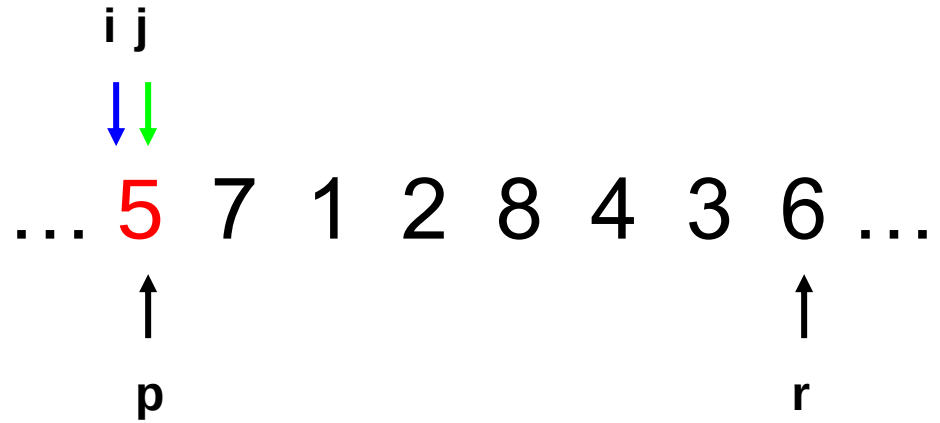



```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

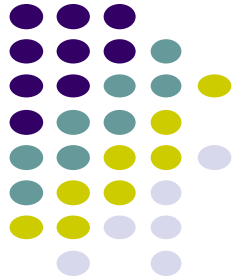


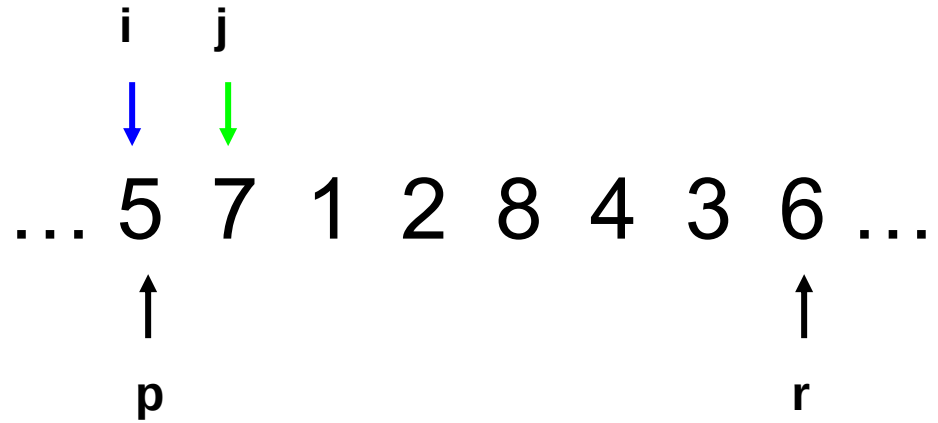

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



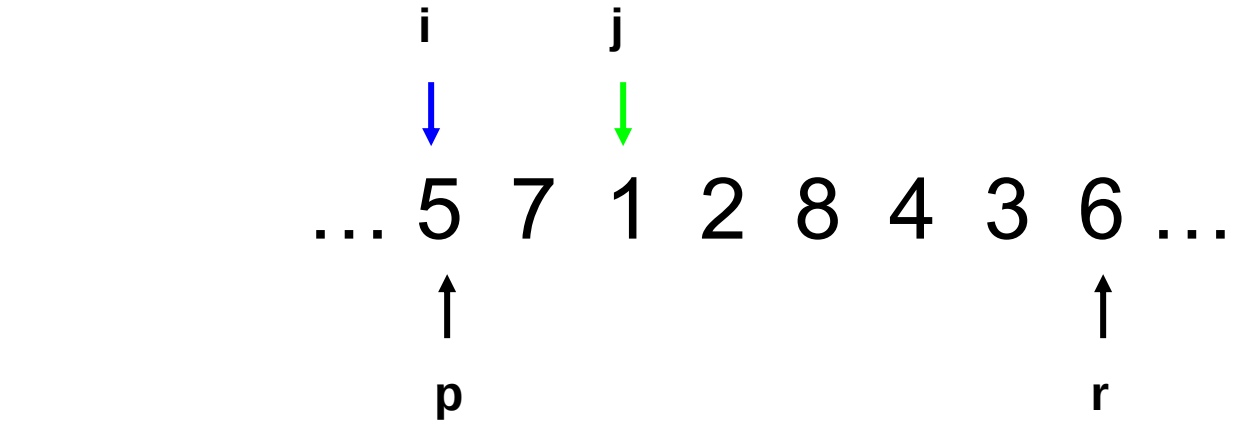



```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

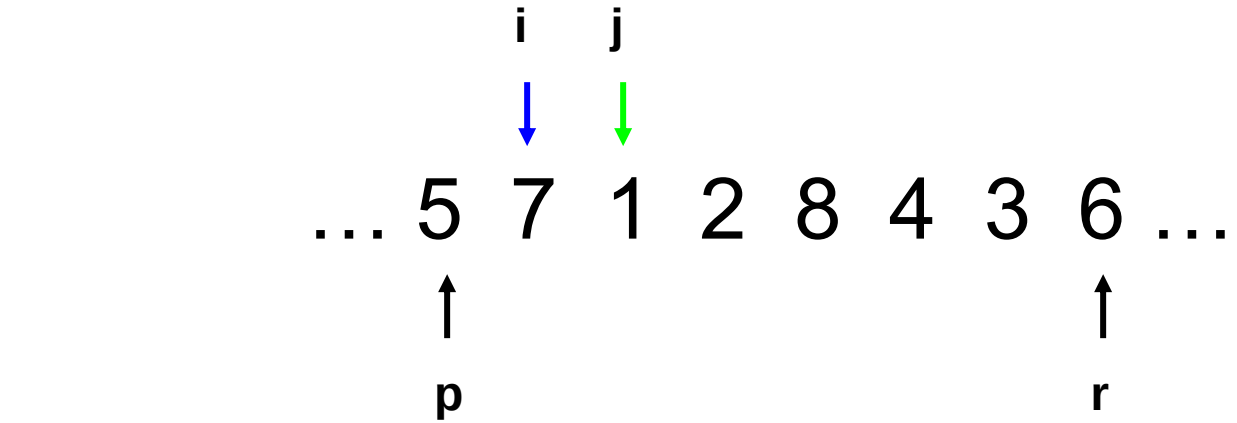




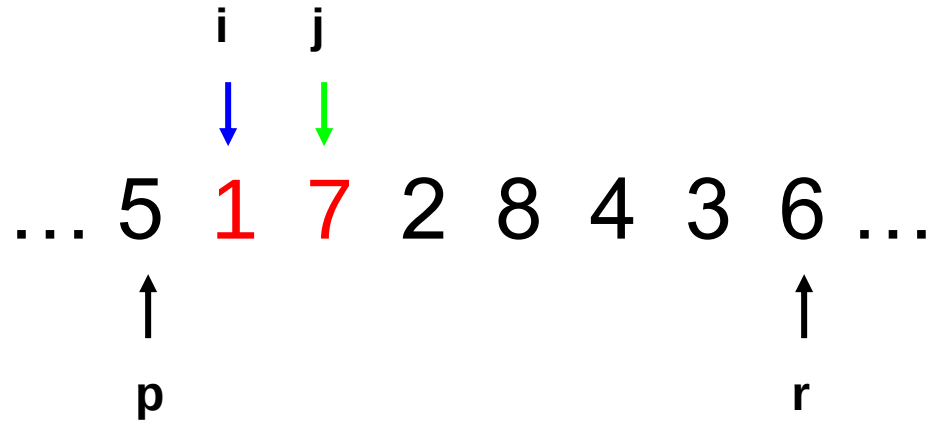

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



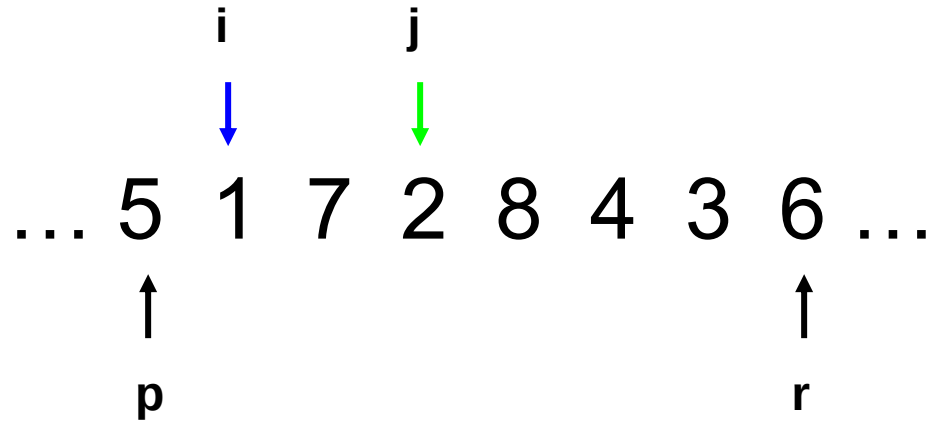

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



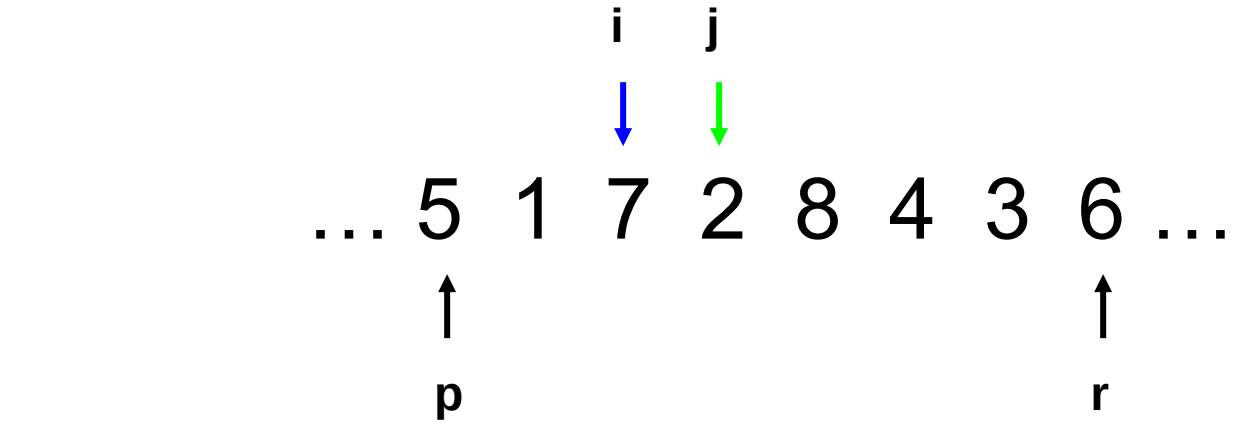

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



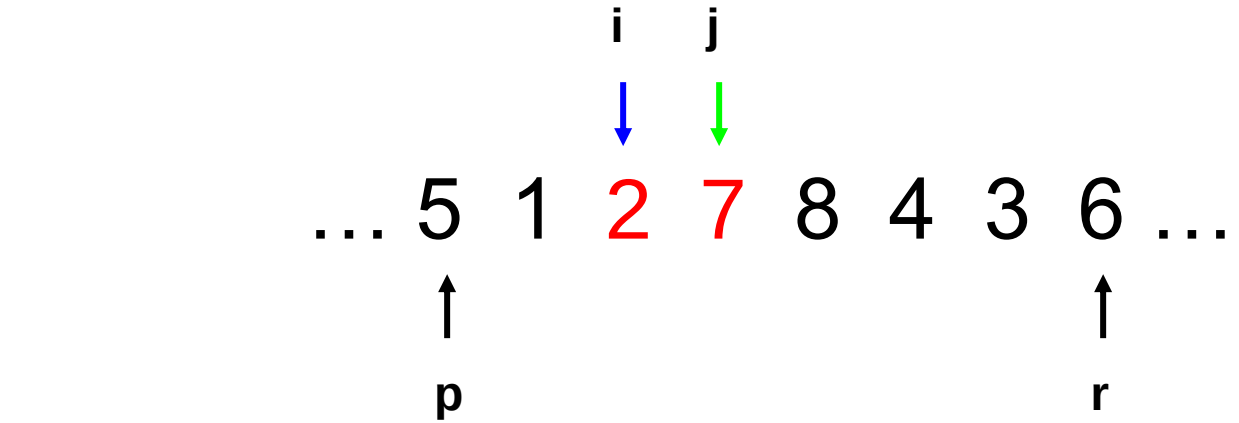

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



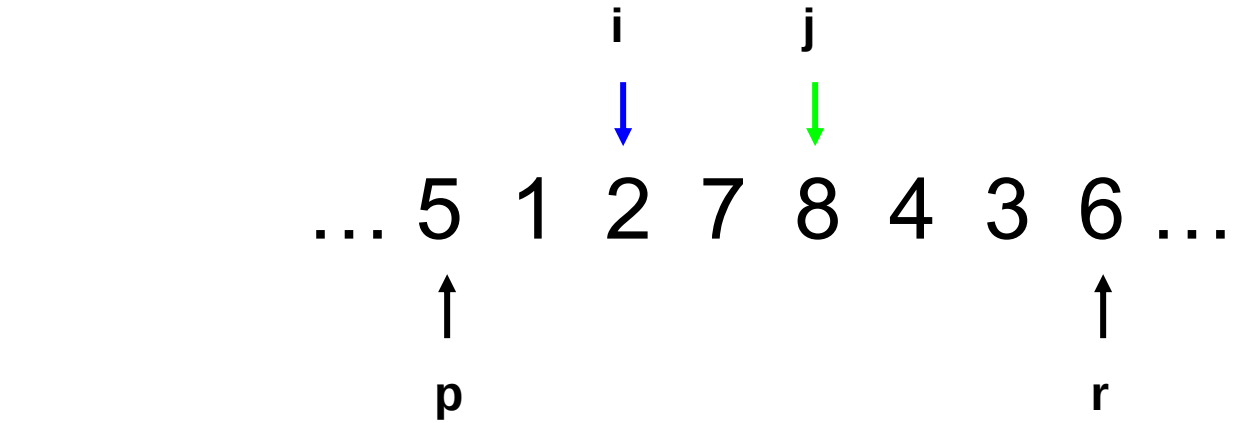

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



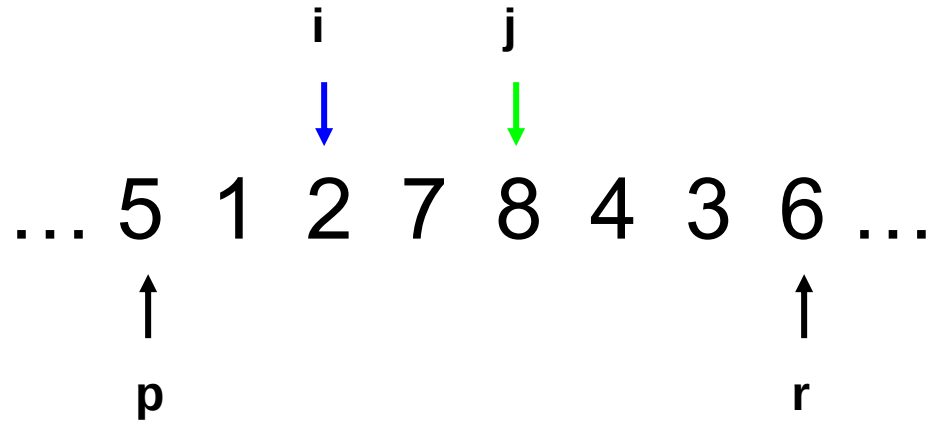

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



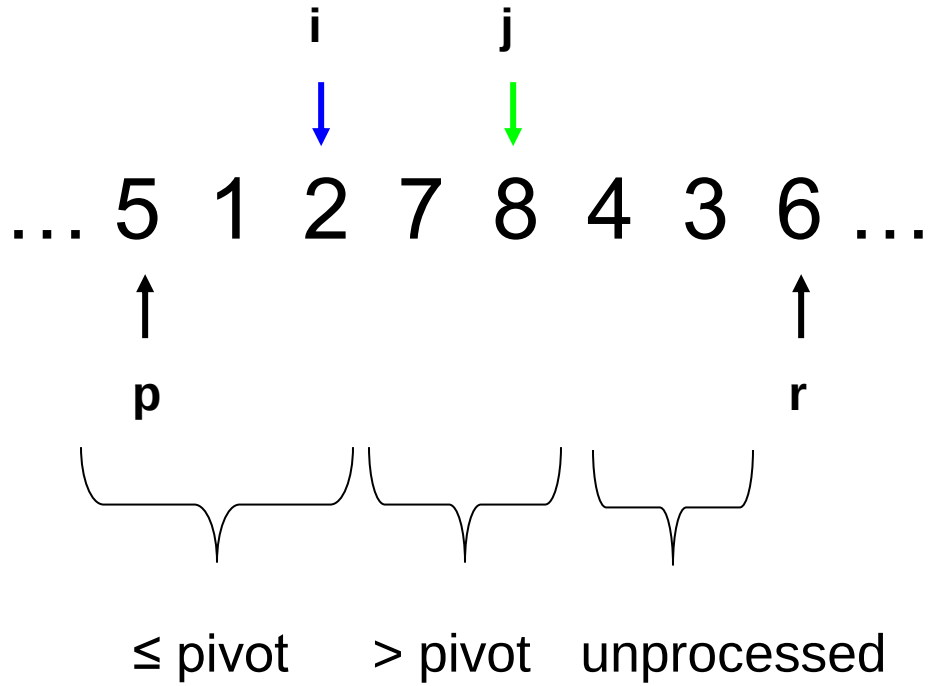

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

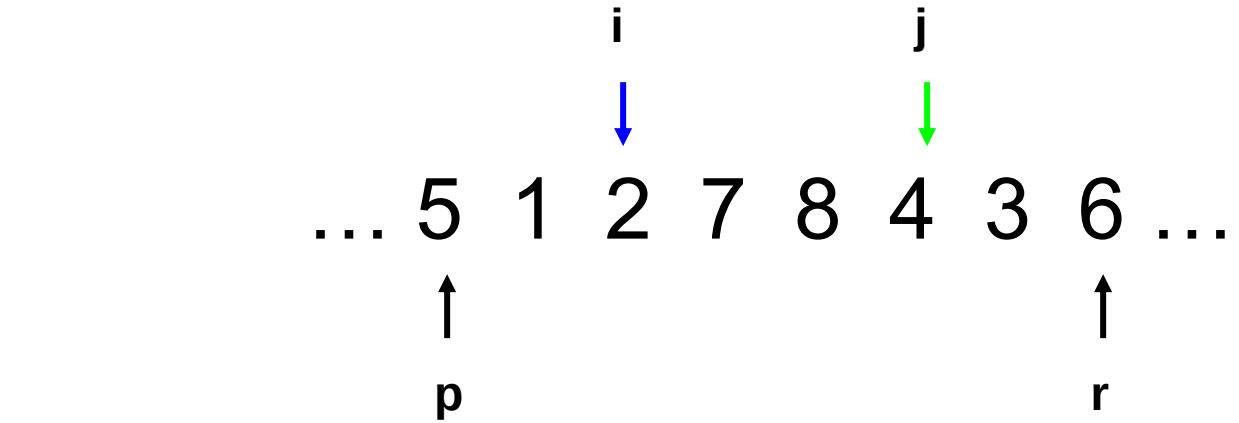



```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

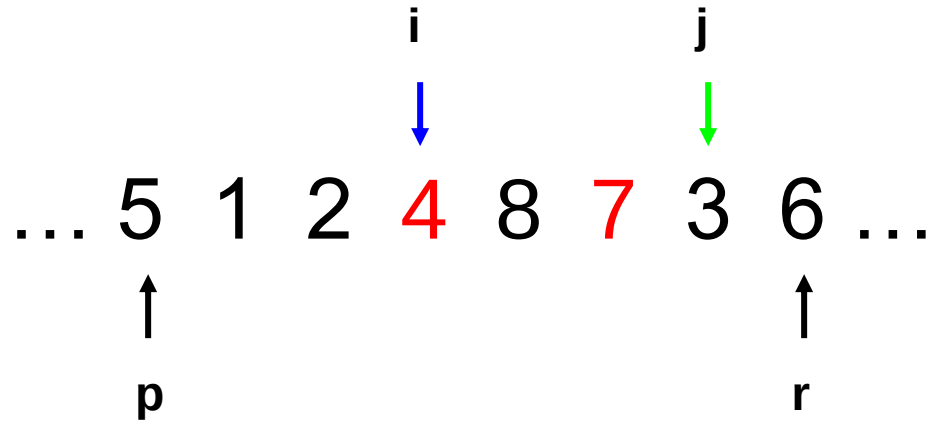


What's happening?

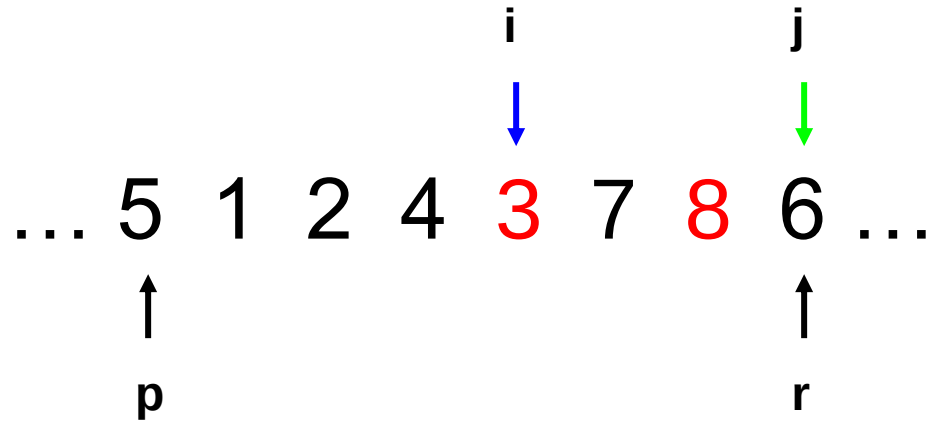

i
j
… 5 1 2 7 8 4 3 6 …
p
r
≤ pivot
> pivot
unprocessed



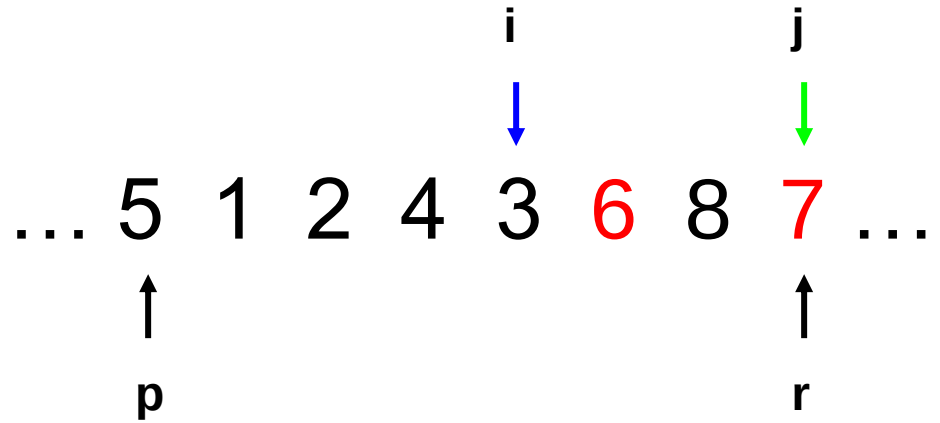

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



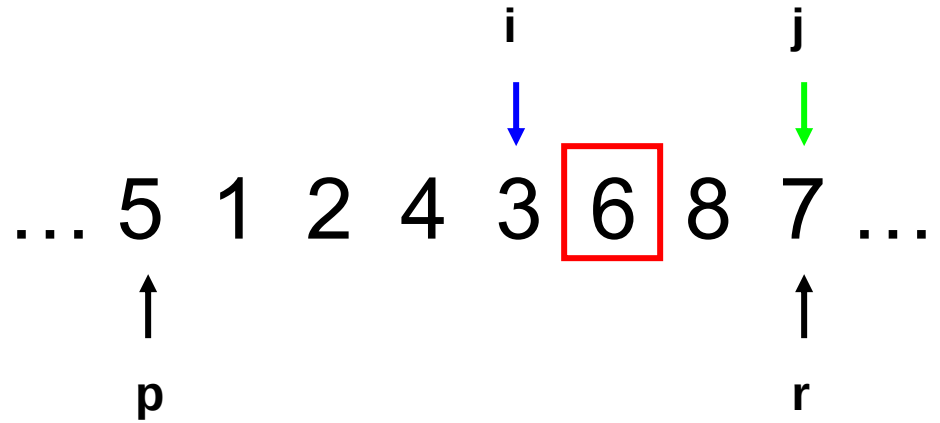

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```




```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```




```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```




```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

Is Partition correct?

Partitions the elements A[p…r-1] in to two sets, those ≤ pivot and those > pivot?

Loop Invariant:

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

Is Partition correct?

Partitions the elements A[p…r-1] in to two sets, those ≤ pivot and those > pivot?

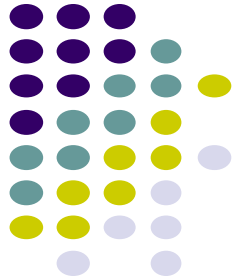
Loop Invariant:

A[p…i] ≤ A[r] and A[i+1…j-1] > A[r]

proof?

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

Proof by induction



Loop Invariant: A[p…i] ≤ A[r] and A[i+1…j-1] > A[r]

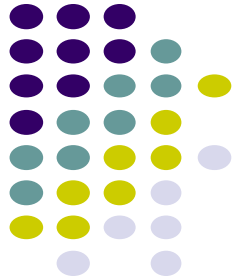
Base case: A[p…i] and A[i+1…j-1] are empty

Assume it holds for *j* -1, two cases:

- A[j] > A[r]
  - A[p…i] remains unchanged
  - A[i+1…j] contains one additional element, A[j] which is > A[r]

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

Proof by induction



Loop Invariant: A[p…i] ≤ A[r] and A[i+1…j-1] > A[r]

2nd case:

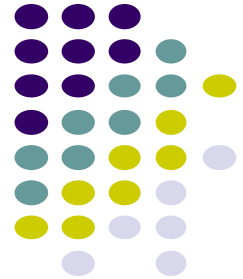
- A[j] ≤ A[r]
  - *i* is incremented
  - A[i] swapped with A[j] – A[p…i] constains one additional element which is ≤ A[r]
  - A[i+1…j-1] will contain the same elements, except the last element will be the old first element

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3          if A[j] ≤ A[r]
4                  i ← i + 1
5                  swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```

Partition running time?

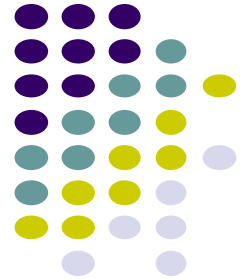
Θ(n)

```
PARTITION(A, p, r)
1  i ← p − 1
2  for j ← p to r − 1
3        if A[j] ≤ A[r]
4              i ← i + 1
5              swap A[i] and A[j]
6  swap A[i + 1] and A[r]
7  return i + 1
```



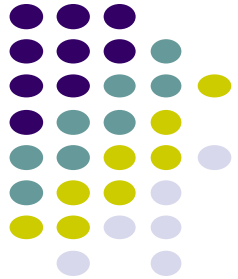
8	5	1	3	6	2	7	4

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



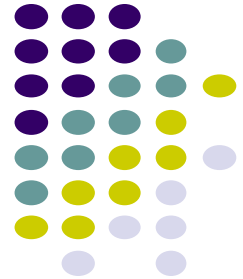
8	5	1	3	6	2	7	4

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



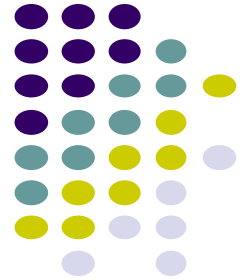
1	3	2	4	6	8	7	5

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



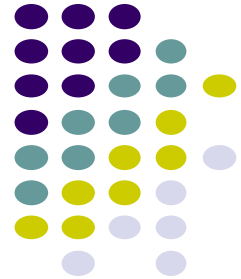
1 3 2 4 6 8 7 5

```
QUICKSORT(A, p, r)
1  if p < r
2          q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```



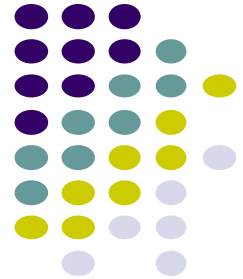
1 3 2 4 6 8 7 5

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



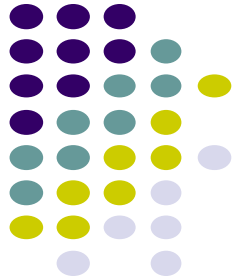
1 2 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



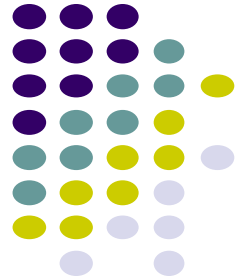
1 2 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
1  if p < r
2          q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```



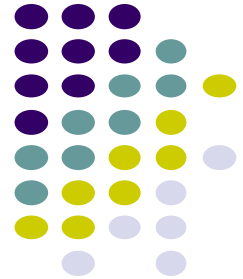
1 2 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



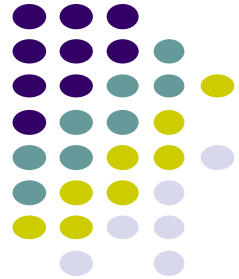
1 2 3 4 6 8 7 5

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



1 2 3 4 6 8 7 5

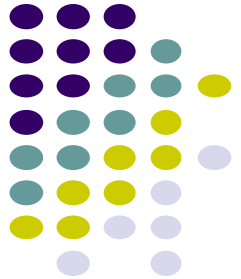
```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



1 2 3 4 5 8 7 6

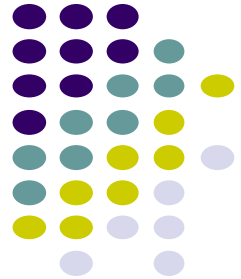
What happens here?

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



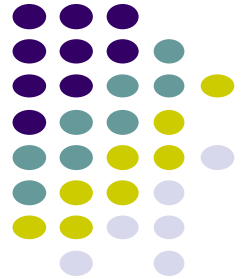
1 2 3 4 5 8 7 6

```
QUICKSORT(A, p, r)
1  if p < r
2          q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```



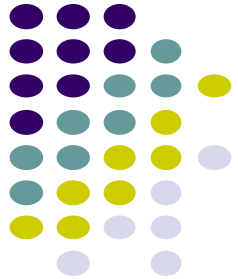
1 2 3 4 5 8 7 6

```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



1 2 3 4 5 6 7 8

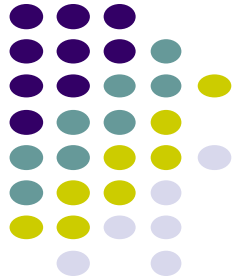
```
QUICKSORT(A, p, r)
1  if p < r
2        q ← PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```



1 2 3 4 5 6 7 8

```
QUICKSORT(A, p, r)
1  if p < r
2          q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

Some observations



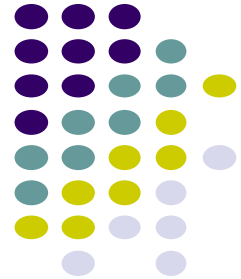
Divide and conquer: different than MergeSort – does the work before recursing

How many times is/can an element selected for as a pivot?

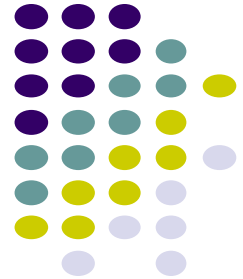
What happens after an element is selected as a pivot?

1	3	2	4	6	8	7	5

Is Quicksort correct?



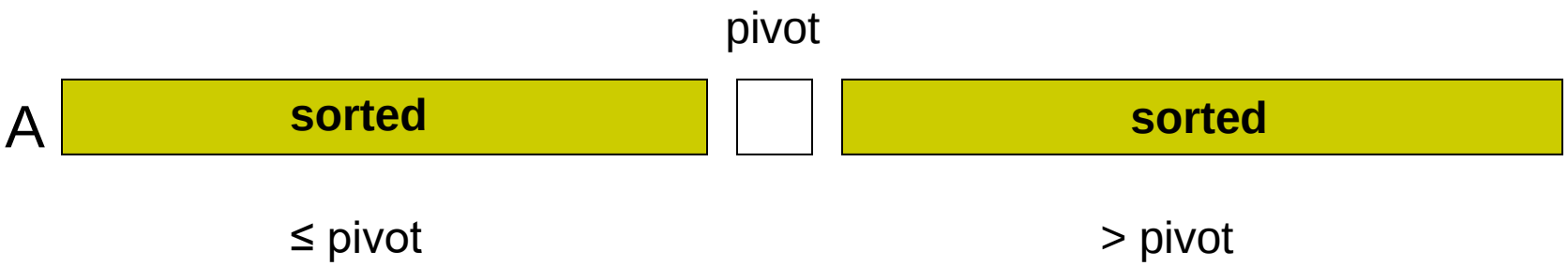
Is Quicksort correct?



Assuming Partition is correct

Proof by induction

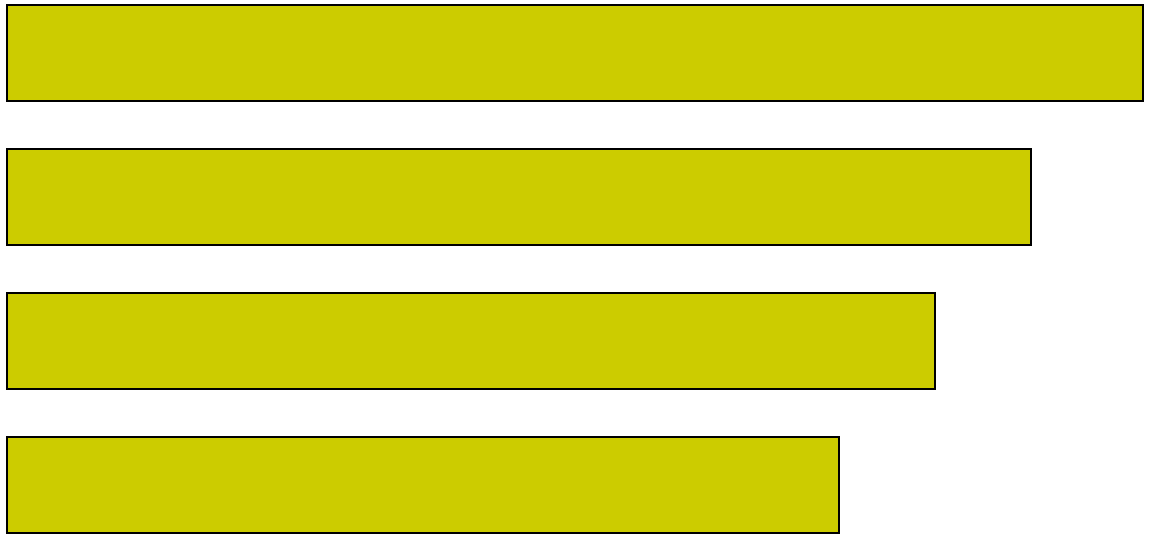
- Base case: Quicksort works on a list of 1 element
- Inductive case:
  - Assume Quicksort sorts arrays for arrays of smaller < n elements, show that it works to sort n elements
  - If partition works correctly then we have:
  - and, by our inductive assumption, we have:



Running time of Quicksort?

Worst case?

Each call to Partition splits the array into an empty array and n-1 array



Quicksort: Worse case running time

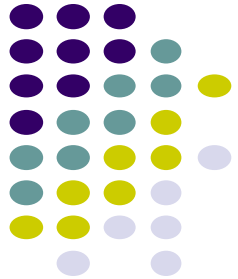
$$T(n) = T(n-1) + \Theta(n)$$

Which is? $\Theta(n^2)$

When does this happen?

- sorted
- reverse sorted
- near sorted/reverse sorted

Quicksort best case?



Each call to Partition splits the array into two equal parts

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

Θ(n log n)

When does this happen?

- Random Data?

Quicksort Average case?

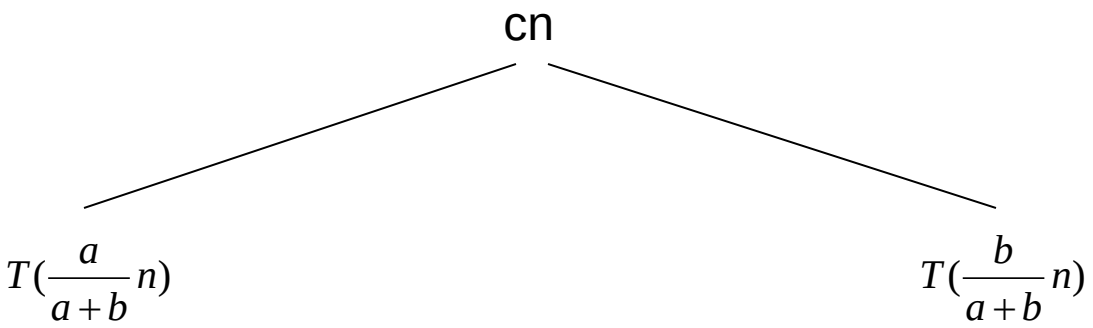
How close to “even” splits do they need to be to maintain an Θ(n log n) running time?

Say the Partition procedure always splits the array into some constant ratio b-to-a, e.g. 9-to-1

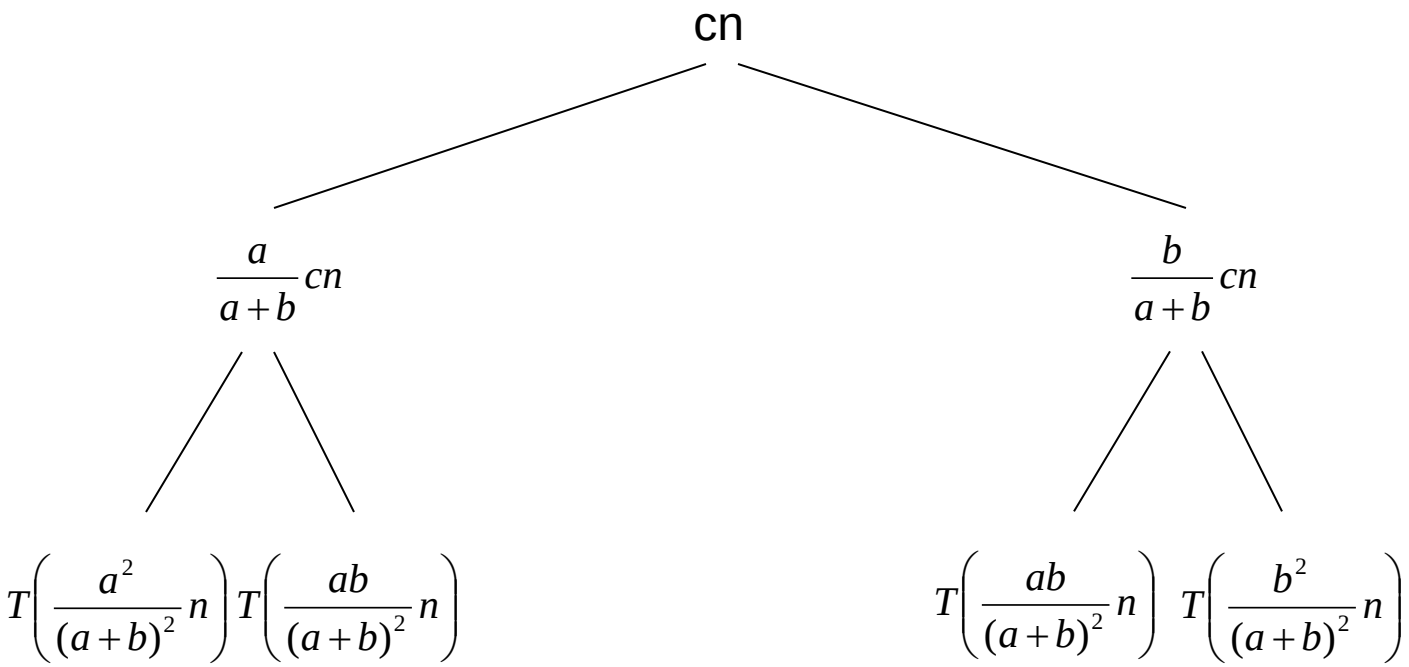
What is the recurrence?

$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$

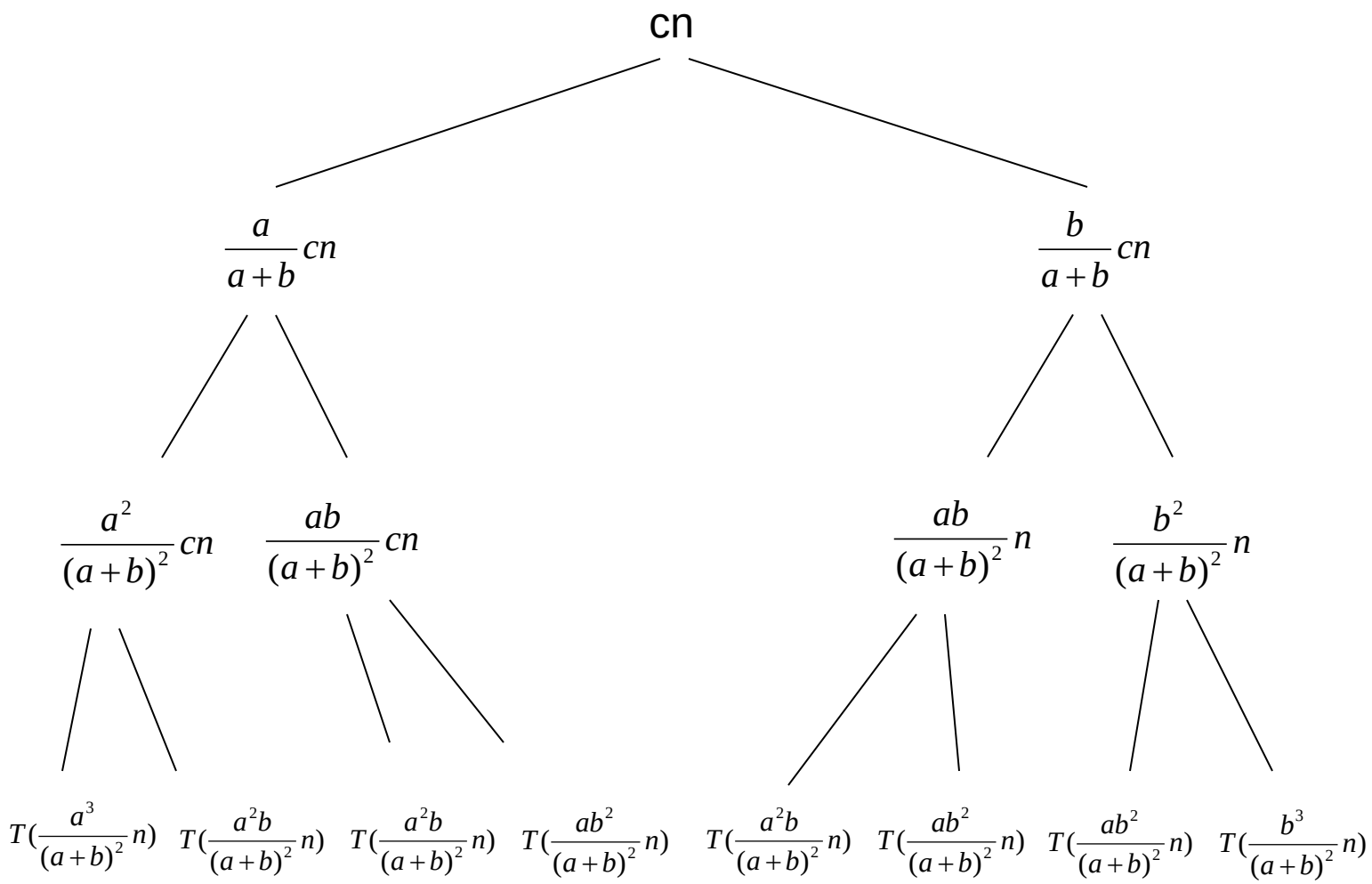
$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$



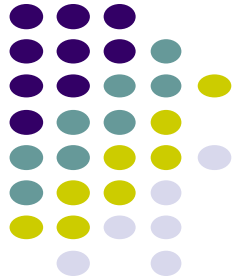
$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$



$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$



$$T(n) \le T\left(\frac{a}{a+b}n\right) + T\left(\frac{b}{a+b}n\right) + cn$$



Level 0: cn

Level 1: $= cn\left(\frac{a}{a+b}\right) + cn\left(\frac{b}{a+b}\right) = cn$

Level 2: $= cn\left(\frac{a^2}{(a+b)^2}\right) + cn\left(\frac{ab}{(a+b)^2}\right) + cn\left(\frac{ab}{(a+b)^2}\right) + cn\left(\frac{b^2}{(a+b)^2}\right)$

$= cn\left(\frac{a^2+2ab+b^2}{(a+b)^2}\right) = cn\left(\frac{(a+b)^2}{(a+b)^2}\right) = cn$

Level 3: $= cn\left(\frac{(a+b)^2 a + (a+b)^2 b}{(a+b)^3}\right)$

$= cn\left(\frac{(a+b)(a+b)^2}{(a+b)^3}\right) = cn$

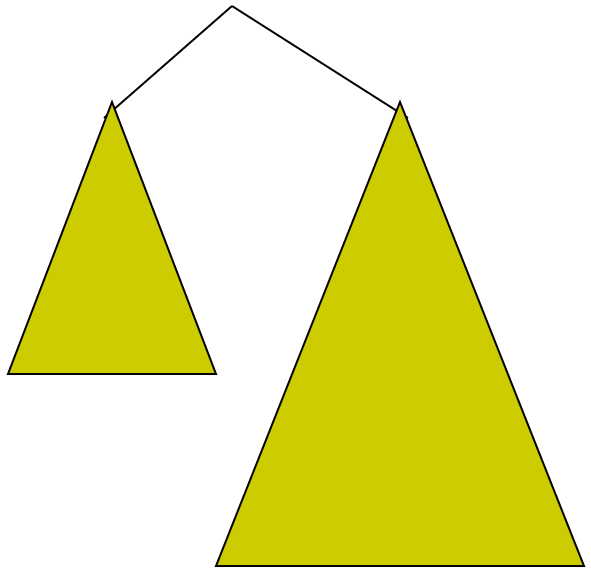
Level d: $= cn\left(\frac{(a+b)^d}{(a+b)^d}\right) = cn$

What is the depth of the tree?

Leaves will have different heights

Want to pick the deepest leave

Assume a < b



What is the depth of the tree?

Assume a < b

$$\left(\frac{b}{a+b}\right)^{d} n = 1$$

...

$$d = \log_{\frac{a+b}{b}} n$$

Cost of the tree

Cost of each level ≤ cn

?

Cost of the tree

Cost of each level ≤ cn

Times the maximum depth

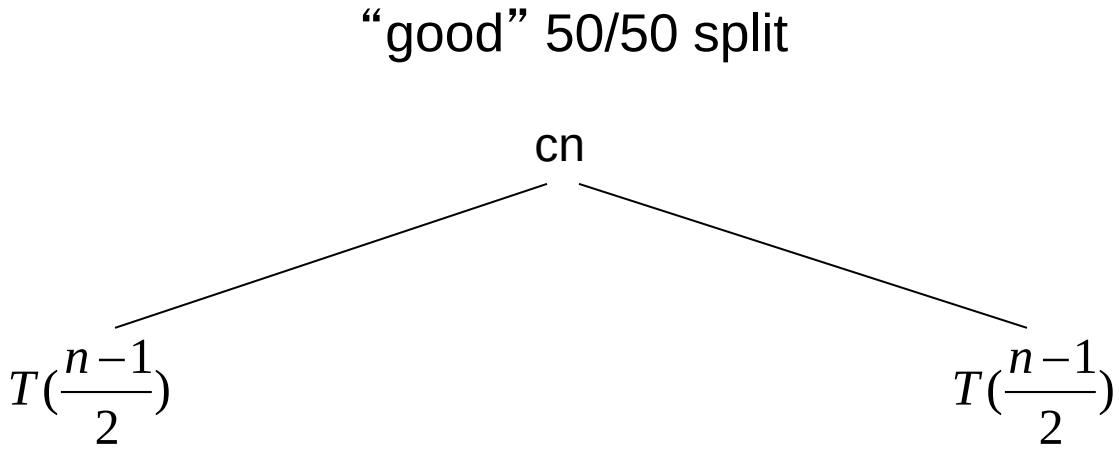
$$O(n \log_{\frac{a+b}{b}} n)$$

Why not?

$$\Theta(n \log_{\frac{a+b}{b}} n)$$

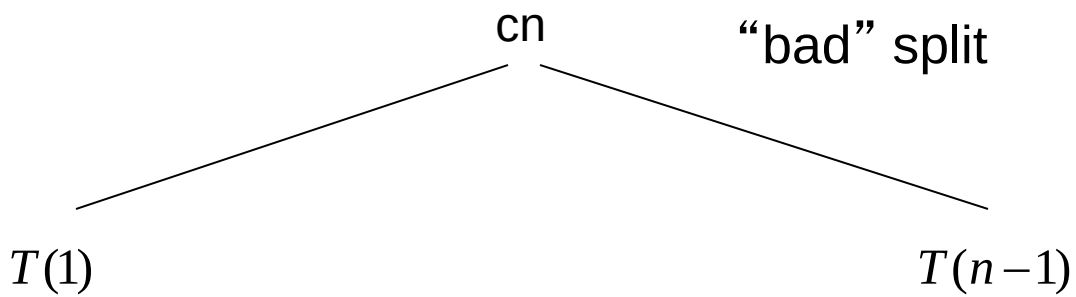
Quicksort average case: take 2

What would happen if half the time Partition produced a "bad" split and the other half "good"?

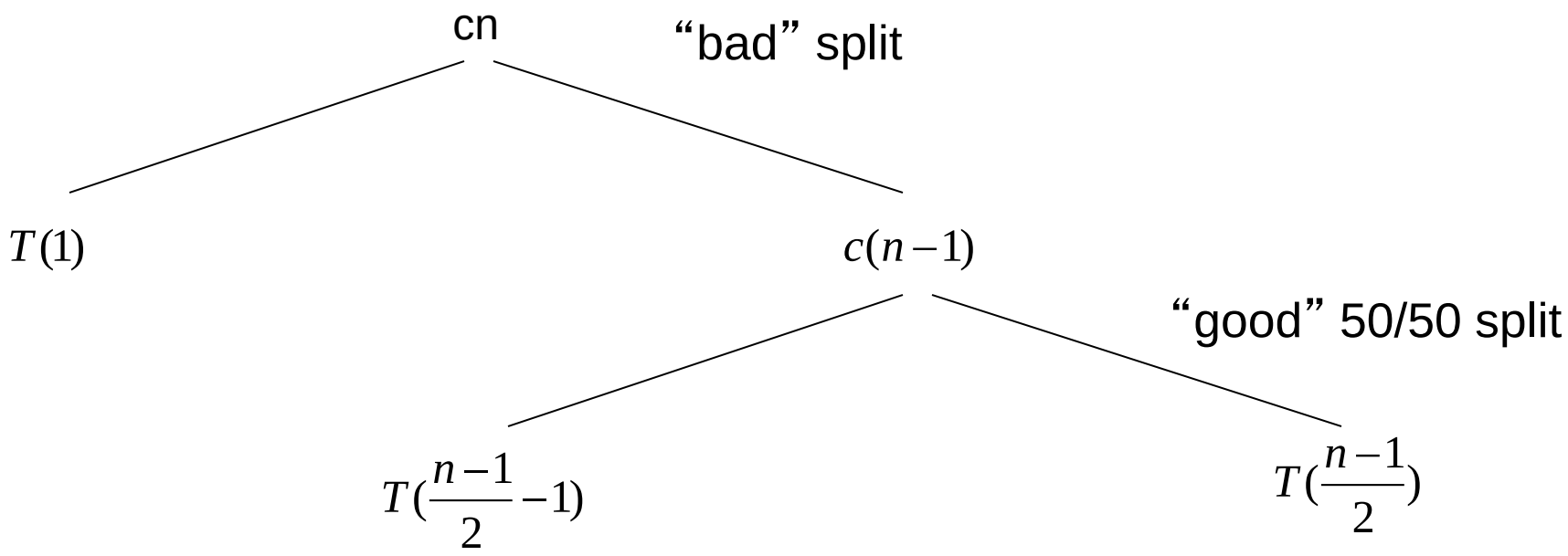


$$T(n) = 2T(\frac{n-1}{2}) + \Theta(n)$$

Quicksort average case: take 2



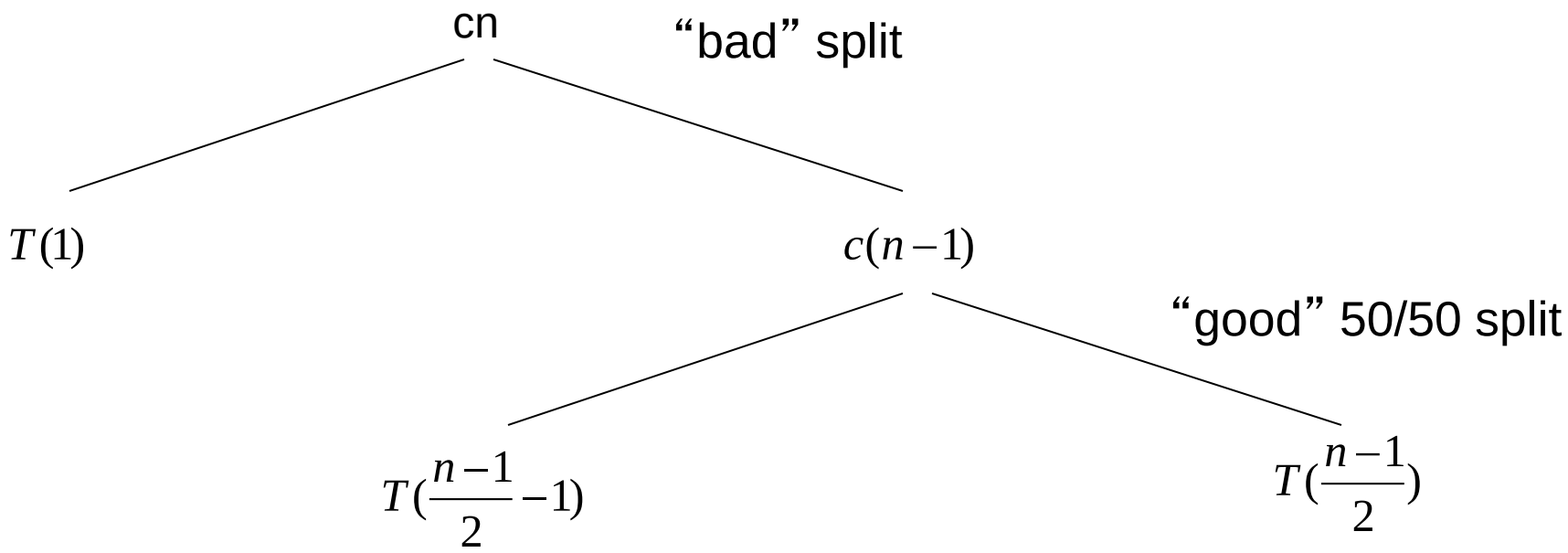
Quicksort average case: take 2



recursion cost partition cost

$$T(n) = T(1) + T(\frac{n-1}{2}-1) + T(\frac{n-1}{2}) + \Theta(n) + \Theta(n-1)$$

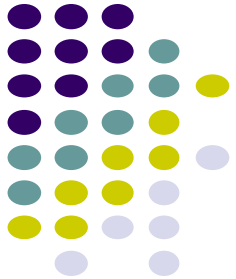
Quicksort average case: take 2



$$T(n) = T(\frac{n-1}{2}-1) + T(\frac{n-1}{2}) + \Theta(n)$$

We absorb the “bad” partition. In general, we can absorb any constant number of “bad” partitions

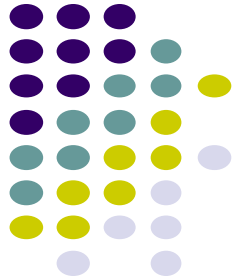
How can we avoid the worst case?



Inject randomness into the data

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  swap A[r] and A[i]
3  returnPARTITION(A, p, r)
```

What is the running time of randomized Quicksort?



Worst case?

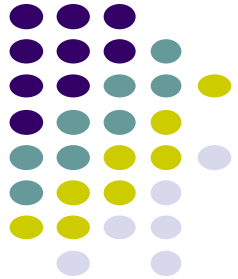
$O(n^2)$

Still could get very unlucky and pick “bad” partitions at every step

Heapsort

- Combines the better attributes of merge sort and insertion sort.
  - Like merge sort, but unlike insertion sort, running time is $O(n \lg n)$.
  - Like insertion sort, but unlike merge sort, sorts in place.
- Introduces an algorithm design technique
  - Create data structure (*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - Priority Queues

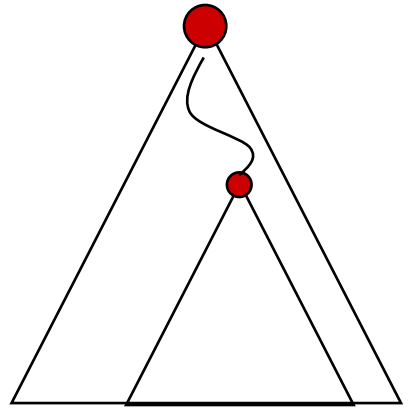
Data Structure Binary Heap



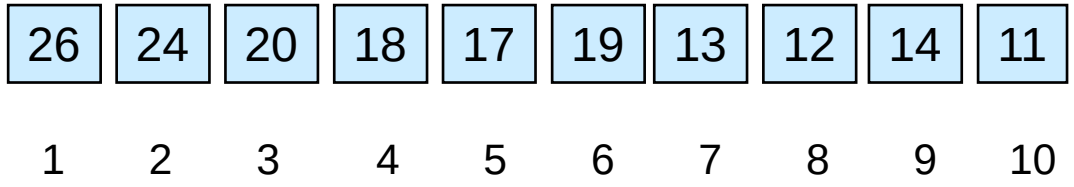
- Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)
- Map from array elements to tree nodes and vice versa
  - Root – $A[1]$
  - Left[i] – $A[2i]$
  - Right[i] – $A[2i+1]$
  - Parent[i] – $A[\lfloor i/2 \rfloor]$
- length[A] – number of elements in array A.
- heap-size[A] – number of elements in heap stored in A.
  - heap-size[A] $\leq$ length[A]

Heap Property (Max and Min)

- Max-Heap
  - For every node excluding the root,
    value is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.
- Min-Heap
  - For every node excluding the root,
    value is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root

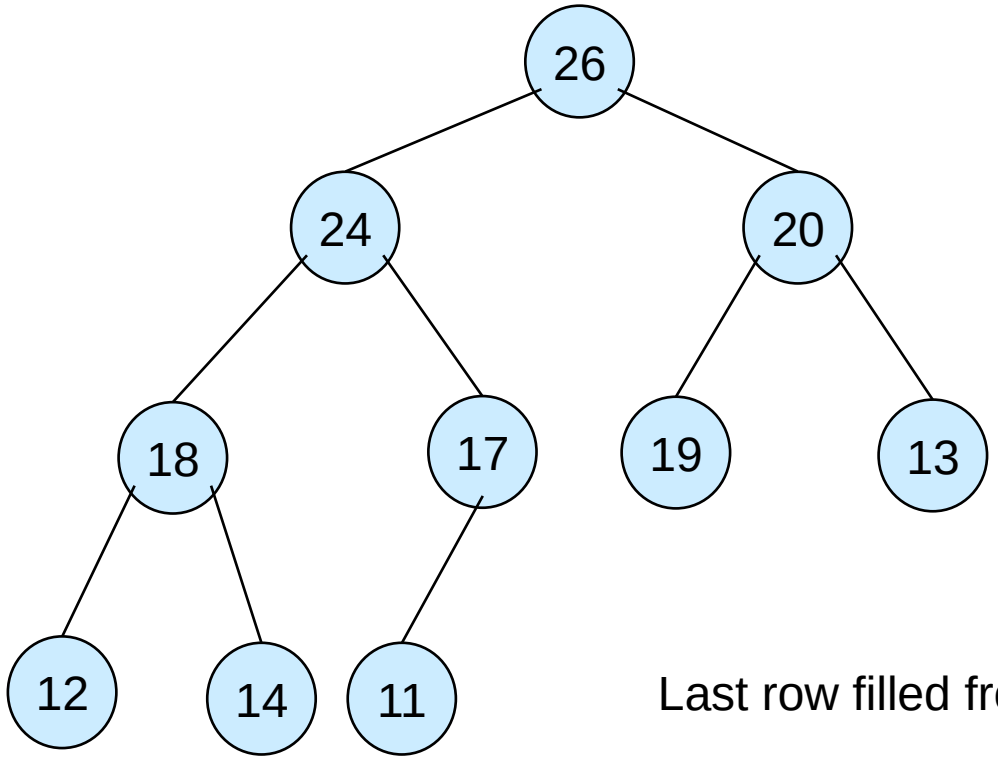


Heaps – Example



Max-heap as an array.

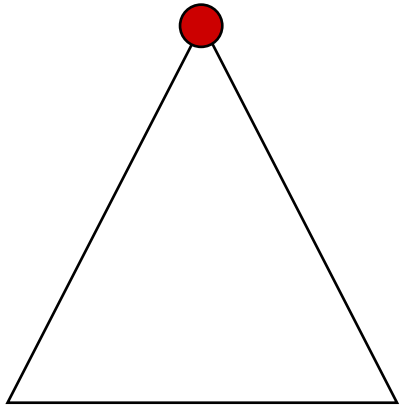
Max-heap as a binary tree.



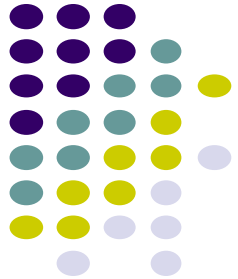
Last row filled from left to right.

Height

- *Height of a node in a tree*: the number of edges on the longest simple downward path from the node to a leaf.
- *Height of a tree*: the height of the root.
- Height of a heap: $\lfloor \lg n \rfloor$
  - Basic operations on a heap run in $O(\lg n)$ time



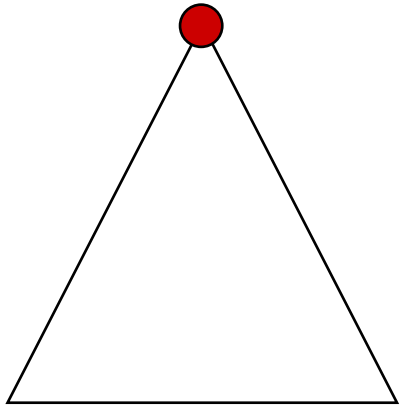
Heaps in Sorting



- Use max-heaps for sorting.
- The array representation of max-heap is not sorted.
- Steps in sorting
  - Convert the given array of size *n* to a max-heap (*BuildMaxHeap*)
  - Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves *n* – 1 elements to be placed in their appropriate locations.
    - However, the array of first *n* – 1 elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (MaxHeapify)
    - Repeat step 2 until the array is sorted.

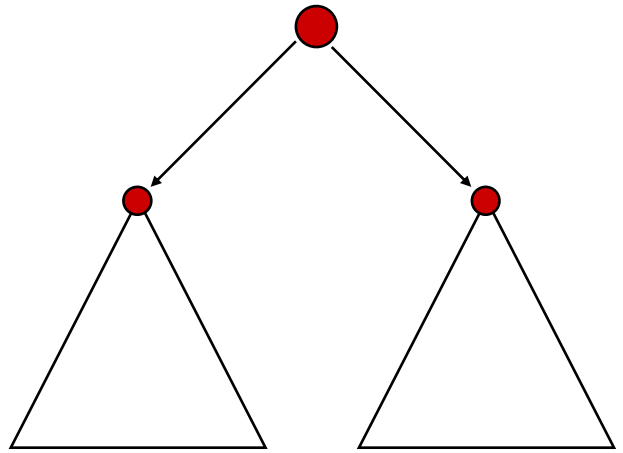
Heap Characteristics

- *Height* $= \lfloor \lg n \rfloor$
- No. of *leaves* $= \lceil n/2 \rceil$
- No. of nodes of height h $\leq \lceil n/2^{h+1} \rceil$



Maintaining the heap property

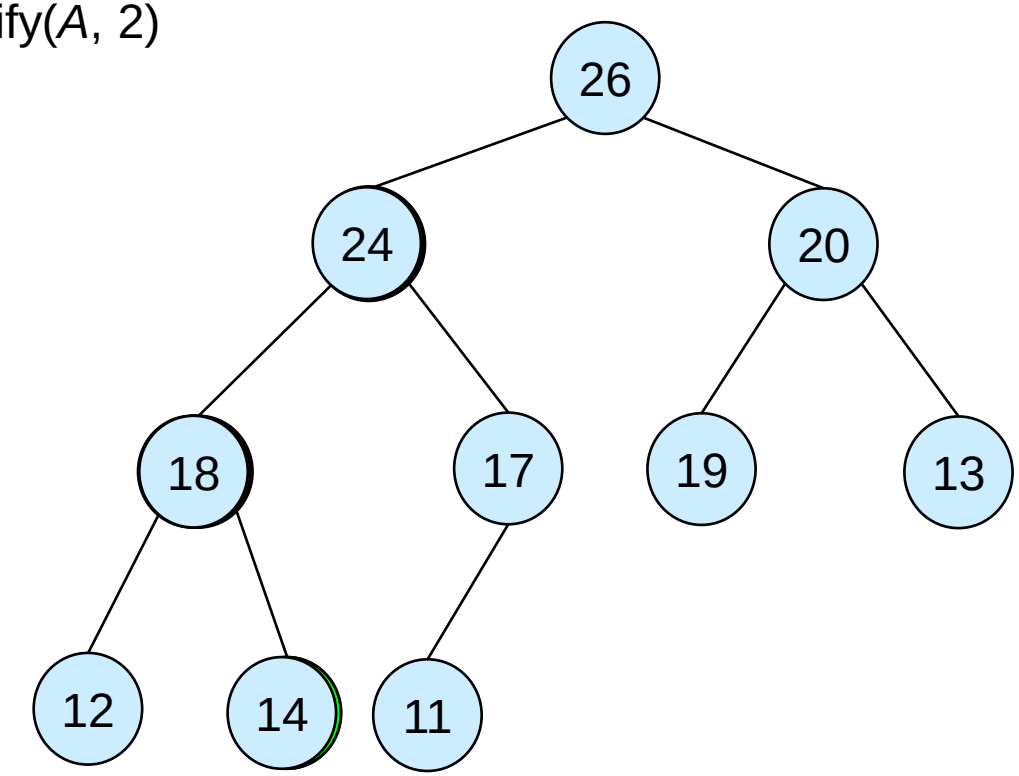
- Suppose two subtrees are max-heaps, but the root violates the max-heap property.



- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - May lead to the subtree at the child not being a heap.
- Recursively fix the children until all of them satisfy the max-heap property.

MaxHeapify – Example

MaxHeapify(*A*, 2)



Procedure MaxHeapify

MaxHeapify(*A*, *i*)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
4. **then** $largest \leftarrow l$
5. **else** $largest \leftarrow i$
6. **if** $r \leq heap\text{-}size[A]$ **and** $A[r] > A[largest]$
7. **then** $largest \leftarrow r$
8. **if** $largest \neq i$
9. **then** exchange $A[i] \leftrightarrow A[largest]$
10. *MaxHeapify*(*A*, *largest*)

Assumption:
Left(*i*) and Right(*i*) are max-heaps.

Running Time for MaxHeapify

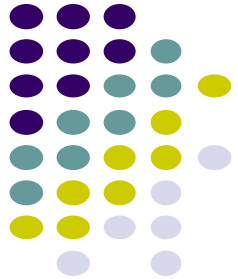
```
MaxHeapify(A, i)
1.  l ← left(i)
2.  r ← right(i)
3.  if l ≤ heap-size[A] and A[l] > A[i]
4.      then largest ← l
5.      else largest ← i
6.  if r ≤ heap-size[A] and A[r] >
    A[largest]
7.      then largest ← r
8.  if largest ≠ i
9.      then exchange A[i] ↔ A[largest]
10.             MaxHeapify(A, largest)
```

Time to fix node *i* and its children
= **Θ(1)**

PLUS

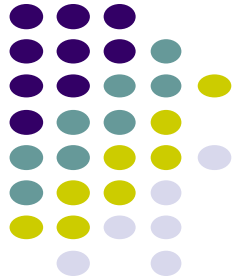
Time to fix the subtree rooted at one of *i*'s children
= *T*(size of subree at *largest*)

Running Time for MaxHeapify(*A*, *n*)



- $T(n) = T(largest) + \Theta(1)$
- $largest \leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$
- Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied

Building a heap

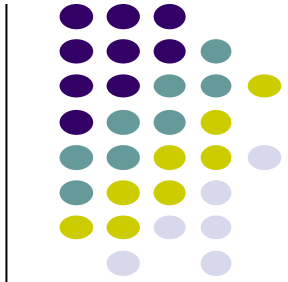


- Use *MaxHeapify* to convert an array *A* into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(*A*)

1. *heap-size*[*A*] ← *length*[*A*]
2. **for** *i* ← $\lfloor$*length*[*A*]/2$\rfloor$ **downto** 1
3. **do** *MaxHeapify*(*A*, *i*)

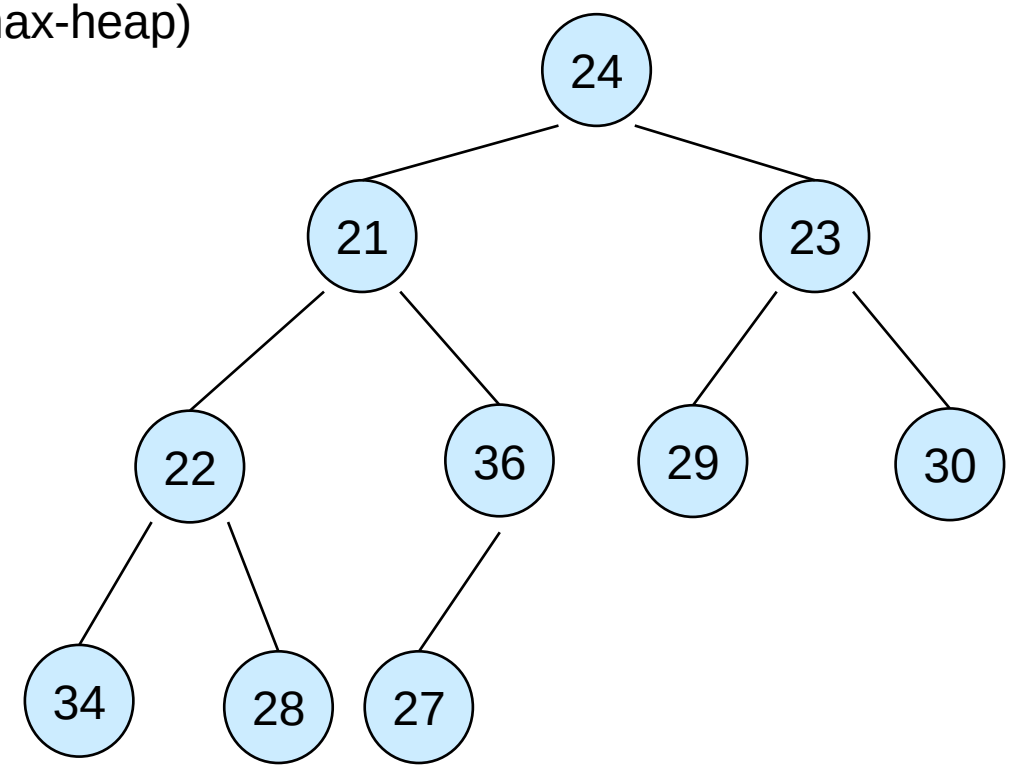
BuildMaxHeap – Example



Input Array:

24	21	23	22	36	29	30	34	28	27

Initial Heap: (not max-heap)



BuildMaxHeap – Example

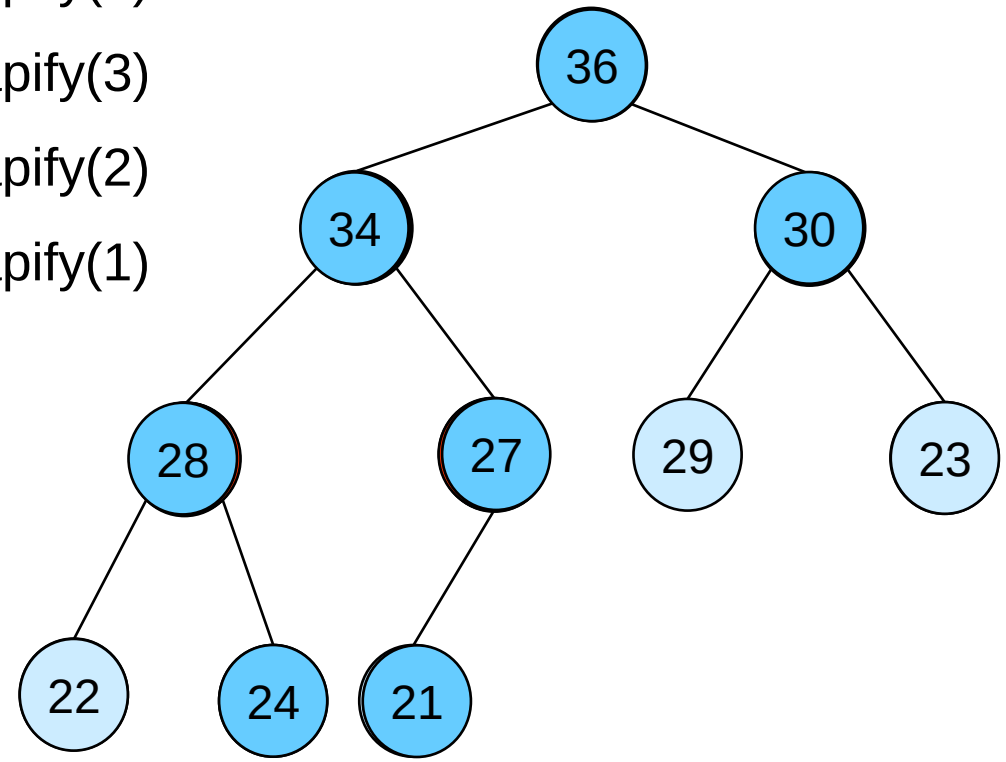
MaxHeapify(⌊10/2⌋ = 5)

MaxHeapify(4)

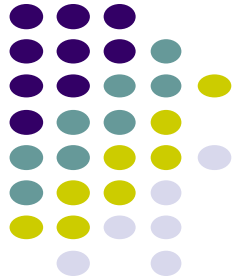
MaxHeapify(3)

MaxHeapify(2)

MaxHeapify(1)

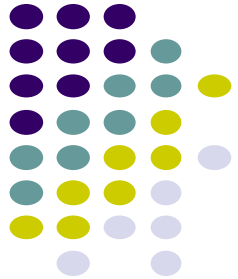


Correctness of *BuildMaxHeap*



- Loop Invariant: At the start of each iteration of the **for** loop, each node i+1, i+2, …, n is the root of a max-heap.
- Initialization:
  - Before first iteration $i = \lfloor n/2 \rfloor$
  - Nodes $\lfloor n/2 \rfloor$+1, $\lfloor n/2 \rfloor$+2, …, n are leaves and hence roots of max-heaps.
- Maintenance:
  - By LI, subtrees at children of node i are max heaps.
  - Hence, MaxHeapify(i) renders node i a max heap root (while preserving the max heap root property of higher-numbered nodes).
  - Decrementing i reestablishes the loop invariant for the next iteration.

Running Time of *BuildMaxHeap*



- Loose upper bound:
  - Cost of a *MaxHeapify* call × No. of calls to *MaxHeapify*
  - $O(\lg n) \times O(n) = O(n \lg n)$
- Tighter bound:
  - Cost of a call to *MaxHeapify* at a node depends on the height, h, of the node – $O(h)$.
  - Height of most nodes smaller than n.
  - Height of nodes h ranges from 0 to $\lfloor \lg n \rfloor$.
  - No. of nodes of height h is $\lceil n/2^{h+1} \rceil$

Running Time of *BuildMaxHeap*

Tighter Bound for *T*(*BuildMaxHeap*)

T(*BuildMaxHeap*)

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}$$

$$\leq \sum_{h=0}^{\infty} \frac{h}{2^h} \qquad , x = 1/2 \text{ in (A.8)}$$

$$= \frac{1/2}{(1-1/2)^2}$$

$$= 2$$

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

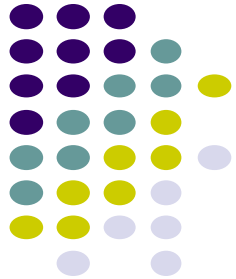
$$= O(n)$$

Can build a heap from an unordered array in linear time

Heapsort

- Sort by maintaining the as yet unsorted elements as a max-heap.
- Start by building a max-heap on all elements in *A.*
  - Maximum element is in the root, *A*[1].
- Move the maximum element to its correct final position.
  - Exchange *A*[1] with *A*[*n*].
- Discard *A*[*n*] – it is now sorted.
  - Decrement heap-size[*A*].
- Restore the max-heap property on *A*[1..*n*–1].
  - Call *MaxHeapify*(*A*, 1).
- Repeat until heap-size[*A*] is reduced to 2.

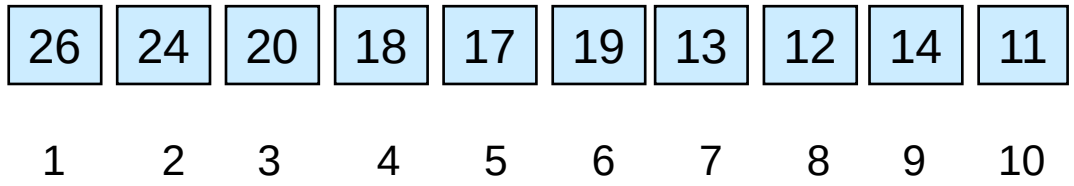
Heapsort(*A*)

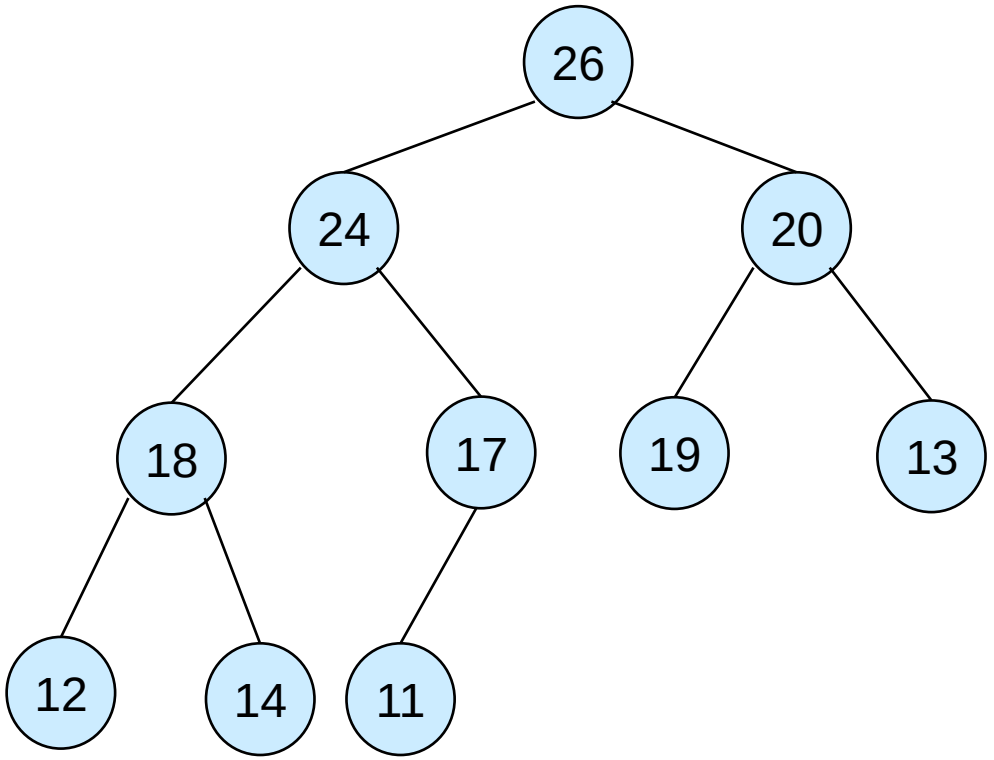


HeapSort(*A*)

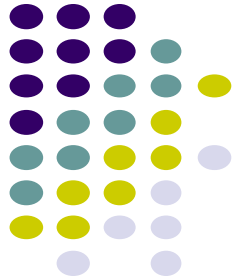
1. Build-Max-Heap(*A*)
2. **for** $i \leftarrow$ *length*[*A*] **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. *heap-size*[*A*] $\leftarrow$ *heap-size*[*A*] – 1
5. *MaxHeapify*(*A*, 1)

Heapsort – Example





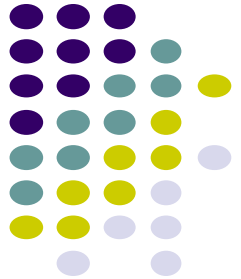
Algorithm Analysis



```
HeapSort(A)
1. Build-Max-Heap(A)
2. for i ← length[A] downto 2
3.       do exchange A[1] ↔ A[i]
4.             heap-size[A] ← heap-size[A] – 1
5.             MaxHeapify(A, 1)
```

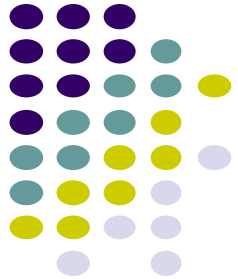
- In-place
- Not Stable
- Build-Max-Heap takes $O(n)$ and each of the $n-1$ calls to Max-Heapify takes time $O(\lg n)$.
- Therefore, $T(n) = O(n \lg n)$

Heap Procedures for Sorting



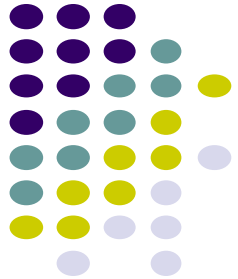
- MaxHeapify $O(\lg n)$
- BuildMaxHeap $O(n)$
- HeapSort $O(n \lg n)$

Priority Queue



- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set *S* of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficiently.
- Applications:
  - Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

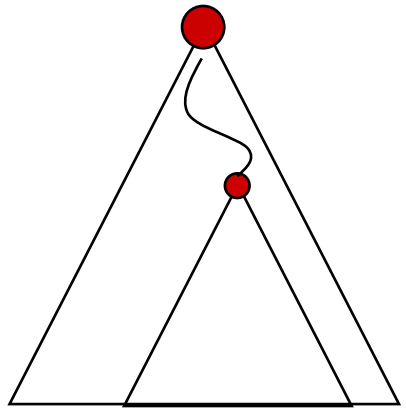
Basic Operations



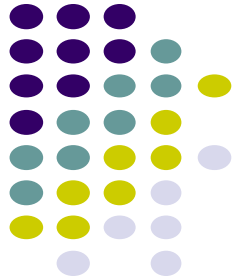
- Operations on a max-priority queue:
  - Insert(S, x) - inserts the element x into the set S
    - $S \leftarrow S \cup \{x\}$.
  - Maximum(S) - returns the element of S with the largest key.
  - Extract-Max(S) - removes and returns the element of S with the largest key.
  - Increase-Key(S, x, k) – increases the value of element x's key to the new value k.
- Min-priority queue supports Insert, Minimum, Extract-Min, and Decrease-Key.
- Heap gives a good compromise between fast insertion but slow extraction and vice versa.

Heap Property (Max and Min)

- Max-Heap
  - For every node excluding the root,
    value is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.
- Min-Heap
  - For every node excluding the root,
    value is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root



Heap-Extract-Max(*A*)

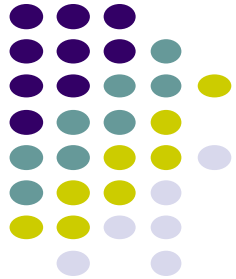


Implements the Extract-Max operation.

Heap-Extract-Max(*A*)
1. if *heap-size*[*A*] < 1
2. then error “heap underflow”
3. *max* ← *A*[1]
4. *A*[1] ← *A*[*heap-size*[*A*]]
5. *heap-size*[*A*] ← *heap-size*[*A*] - 1
6. MaxHeapify(*A*, 1)
7. return max

Running time : Dominated by the running time of MaxHeapify = $O(\lg n)$

Heap-Insert(*A*, *key*)



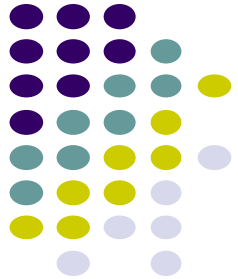
Heap-Insert(*A*, *key*)

1. *heap-size*[*A*] ← *heap-size*[*A*] + 1
2. *i* ← *heap-size*[*A*]
4. **while** *i* > 1 **and** *A*[Parent(*i*)] < *key*
5. **do** *A*[*i*] ← *A*[Parent(*i*)]
6. *i* ← Parent(*i*)
7. *A*[*i*] ← *key*

Running time is $O(\lg n)$

The path traced from the new leaf to the root has length $O(\lg n)$

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



```
Heap-Increase-Key(A, i, key)
1    If key < A[i]
2       then error "new key is smaller than the current key"
3    A[i] ← key
4    while i > 1 and  A[Parent[i]] < A[i]
5          do exchange A[i] ↔ A[Parent[i]]
6                i ← Parent[i]
```

```
Heap-Insert(A, key)
1    heap-size[A] ← heap-size[A] + 1
2    A[heap-size[A]] ← –∞
3    Heap-Increase-Key(A, heap-size[A], key)
```

Examples

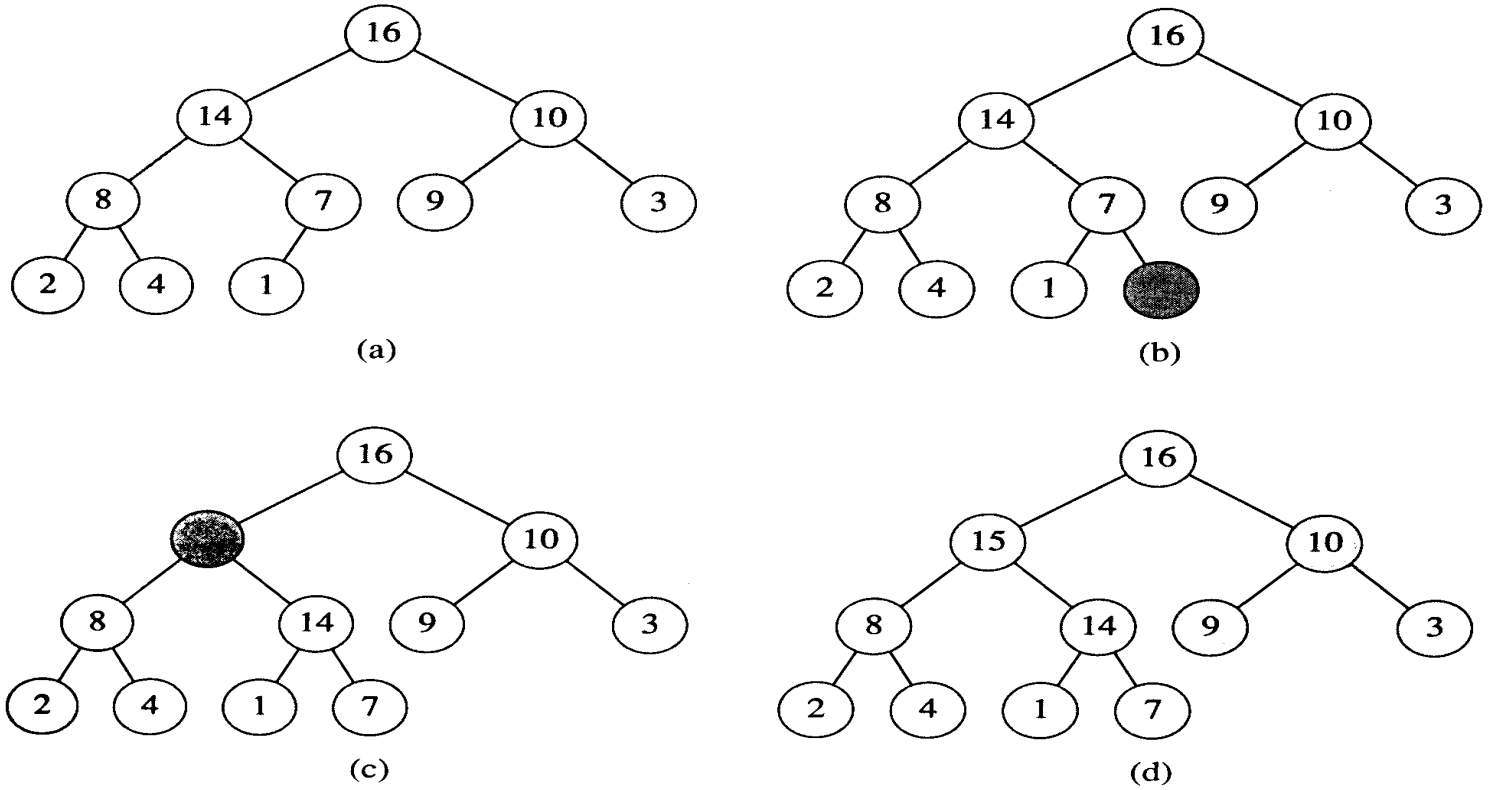


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

Acknowledgements

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., Introduction to algorithms. MIT press, 2009
- Dr. David Kauchak, Pomona College
- Prof. David Plaisted, The University of North Carolina at Chapel Hill