CSE101: Design and Analysis of Algorithms

Ragesh Jaiswal, CSE, UCSD

Greedy Algorithms

- Claim 2: Let S be a subset of vertices containing s such that we know the shortest path length $l(s,u)$ from s to any vertex in $u \in S$. Let $e = (u,v)$ be an edge such that
  1. $u \in S$, $v \in V \setminus S$,
  2. $(l(s,u) + W_e)$ is the least among all such cut edges.

  Then $l(s,v) = l(s,u) + W_e$.

Algorithm

`Dijkstra's Algorithm`(G,s)
- $S \leftarrow \{s\}$
- $d(s) \leftarrow 0$
- While S does not contain all vertices in G
  - Let $e = (u,v)$ be a cut edge across $(S, V \setminus S)$ with minimum value of $d(u) + W_e$
  - $d(v) \leftarrow d(u) + W_e$
  - $S \leftarrow S \cup \{v\}$

- What is the running time of the above algorithm?
  - Same as that of the Prim's algorithm. $O(|E| \cdot \log |V|)$.

Algorithm

`Dijkstra's Algorithm`(G, s)

- $S \leftarrow \{s\}$
- $d(s) \leftarrow 0$
- While S does not contain all vertices in G
  - Let $e = (u, v)$ be a cut edge across $(S, V \setminus S)$ with minimum value of $d(u) + W_e$
  - $d(v) \leftarrow d(u) + W_e$
  - $S \leftarrow S \cup \{v\}$

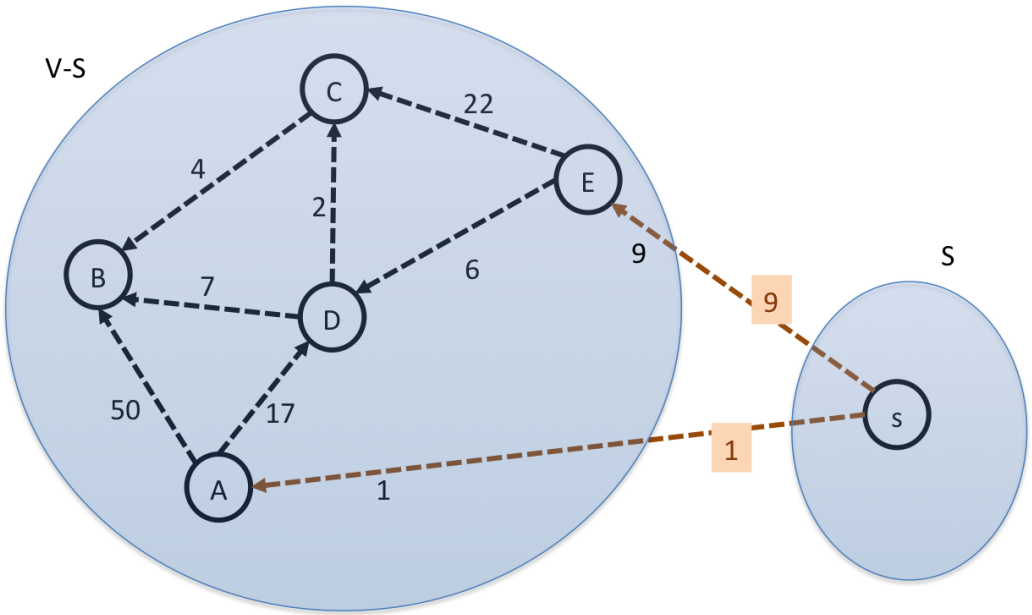


Figure: $d(s) = 0$

Algorithm

`Dijkstra's Algorithm`(G, s)

- $S \leftarrow \{s\}$
- $d(s) \leftarrow 0$
- While S does not contain all vertices in G
  - Let $e = (u, v)$ be a cut edge across $(S, V \setminus S)$ with minimum value of $d(u) + W_e$
  - $d(v) \leftarrow d(u) + W_e$
  - $S \leftarrow S \cup \{v\}$

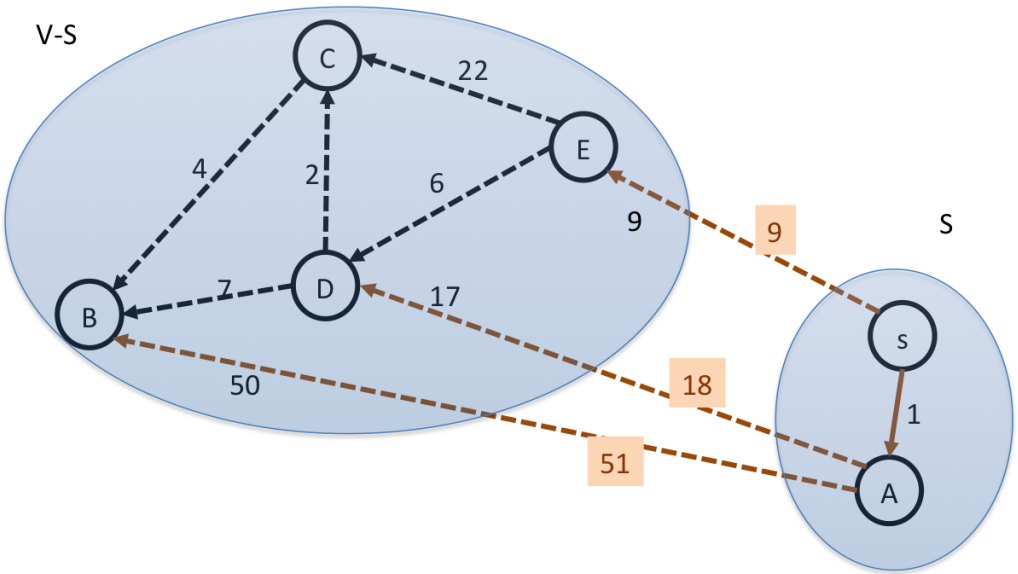


Figure: $d(s) = 0; d(A) = 1$

Algorithm

`Dijkstra's Algorithm`(G, s)

- $S \leftarrow \{s\}$
- $d(s) \leftarrow 0$
- While S does not contain all vertices in G
  - Let $e = (u, v)$ be a cut edge across $(S, V \setminus S)$ with minimum value of $d(u) + W_e$
  - $d(v) \leftarrow d(u) + W_e$
  - $S \leftarrow S \cup \{v\}$

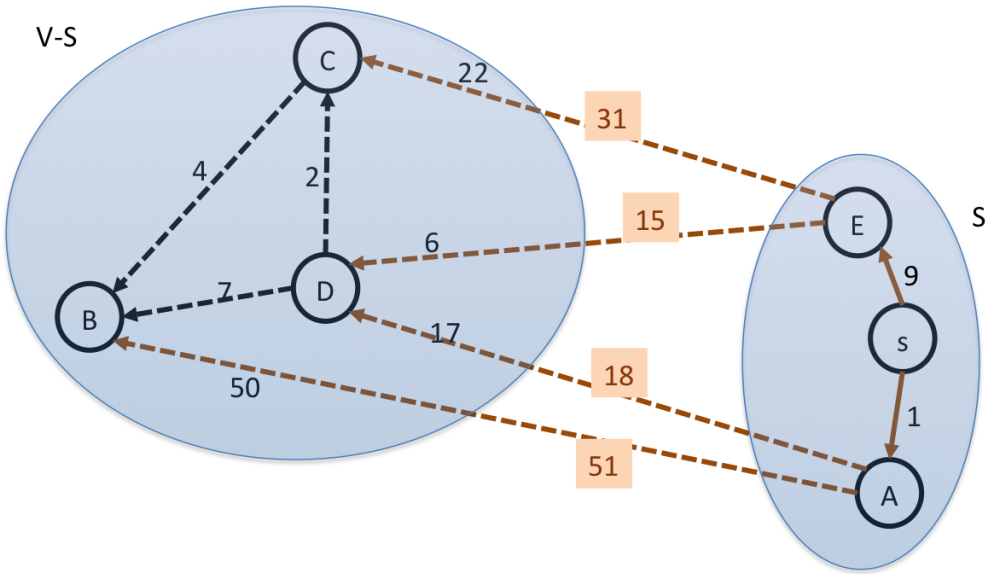


Figure: $d(s) = 0; d(A) = 1; d(E) = 9$

Algorithm

`Dijkstra's Algorithm(`G, s`)`

- $S \leftarrow \{s\}$
- $d(s) \leftarrow 0$
- While S does not contain all vertices in G
  - Let $e = (u, v)$ be a cut edge across $(S, V \setminus S)$ with minimum value of $d(u) + W_e$
  - $d(v) \leftarrow d(u) + W_e$
  - $S \leftarrow S \cup \{v\}$

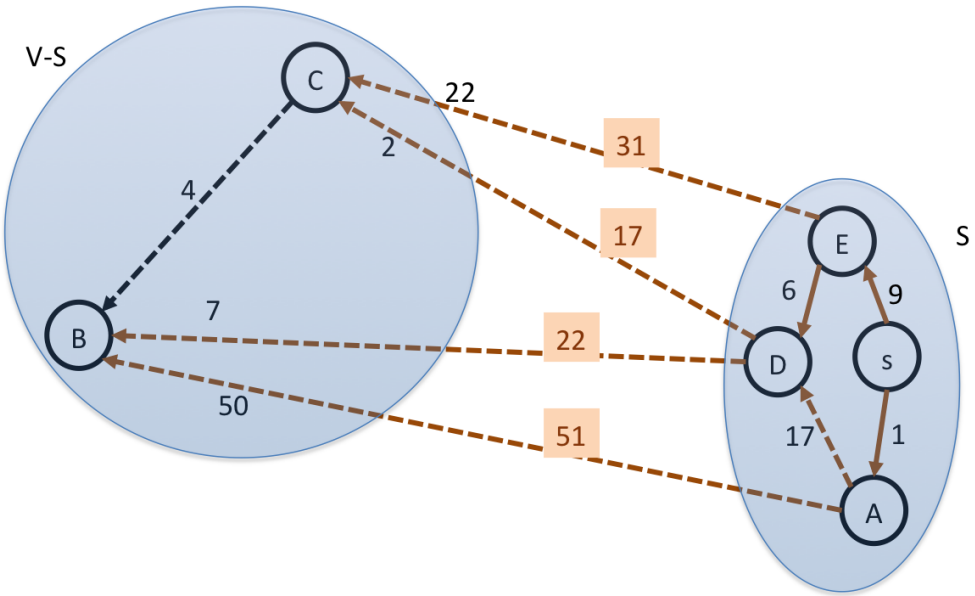


Figure: $d(s) = 0; d(A) = 1; d(E) = 9; d(D) = 15$

Algorithm

`Dijkstra's Algorithm`(G, s)

- $S \leftarrow \{s\}$
- $d(s) \leftarrow 0$
- While S does not contain all vertices in G
  - Let $e = (u, v)$ be a cut edge across $(S, V \setminus S)$ with minimum value of $d(u) + W_e$
  - $d(v) \leftarrow d(u) + W_e$
  - $S \leftarrow S \cup \{v\}$

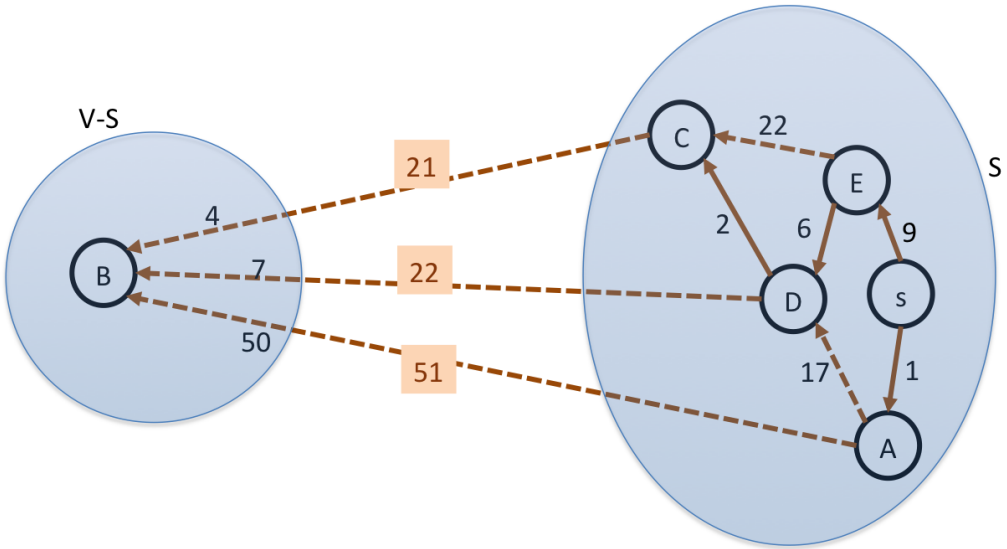


Figure: $d(s) = 0; d(A) = 1; d(E) = 9; d(D) = 15; d(C) = 17$

Algorithm

`Dijkstra's Algorithm`(G, s)

- $S \leftarrow \{s\}$
- $d(s) \leftarrow 0$
- While S does not contain all vertices in G
  - Let $e = (u, v)$ be a cut edge across $(S, V \setminus S)$ with minimum value of $d(u) + W_e$
  - $d(v) \leftarrow d(u) + W_e$
  - $S \leftarrow S \cup \{v\}$

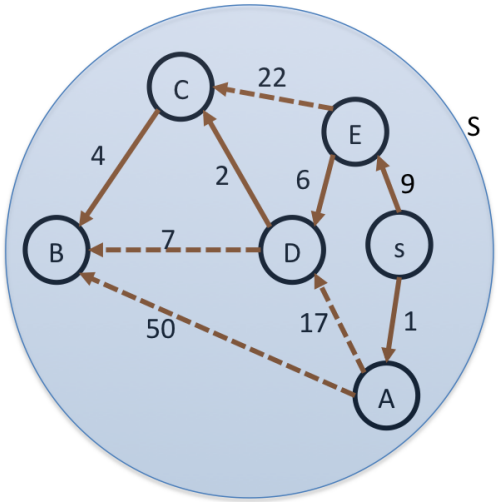


Figure: $d(s) = 0; d(A) = 1; d(E) = 9; d(D) = 15; d(C) = 17; d(B) = 21$

Algorithm

`Dijkstra's Algorithm`(G, s)

- $S \leftarrow \{s\}$
- $d(s) \leftarrow 0$
- While S does not contain all vertices in G
  - Let $e = (u, v)$ be a cut edge across $(S, V \setminus S)$ with minimum value of $d(u) + W_e$
  - $d(v) \leftarrow d(u) + W_e$
  - $S \leftarrow S \cup \{v\}$

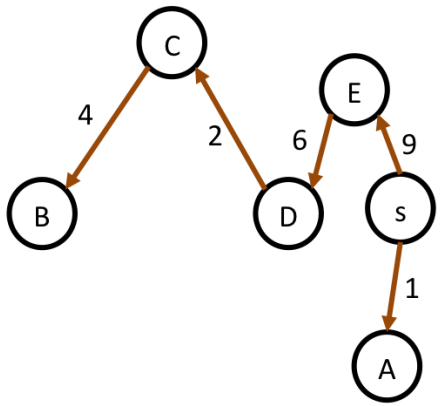


Figure: The algorithm also implicitly produces a *shortest path tree* that gives the shortest paths from s to all vertices.

Course Overview

- Basic graph algorithms
- Algorithm Design Techniques:
  - Greedy Algorithms
  - Divide and Conquer
  - Dynamic Programming
  - Network Flows
- Computational Intractability

Divide and Conquer

- You have already seen multiple examples of Divide and Conquer algorithms:
  - Binary Search
  - Merge Sort
  - Quick Sort
  - Multiplying two n-bit numbers in $O\left(n^{\log_2 3}\right)$ time.

- Main Idea: *Divide the input into smaller parts. Solve the smaller parts and combine their solution.*

Problem

Given an array of unsorted integers, output a sorted array.

Algorithm

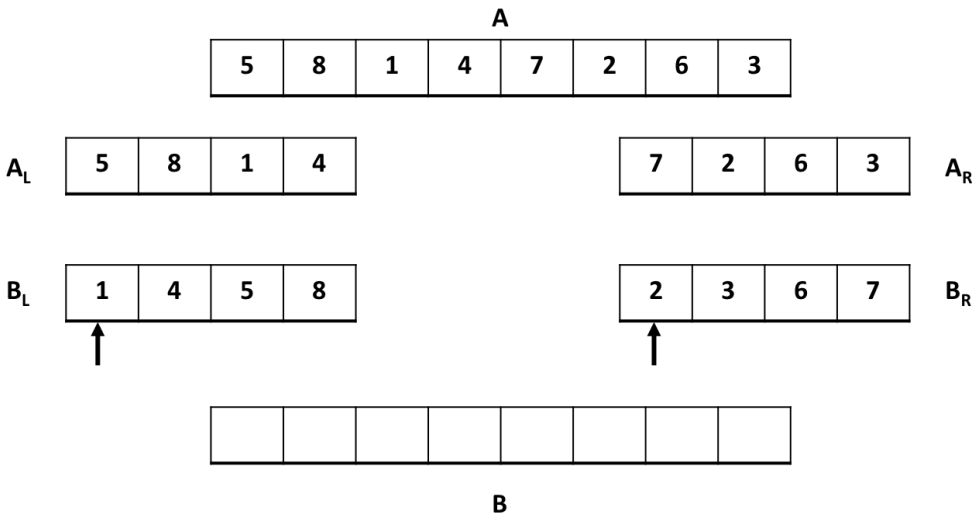
`MergeSort`(A)

- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

Algorithm

`MergeSort`(A)

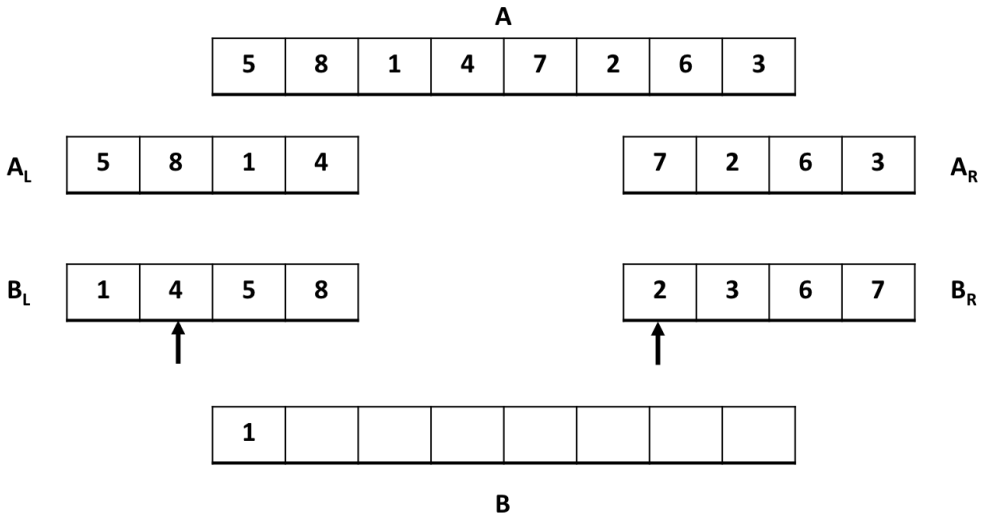
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

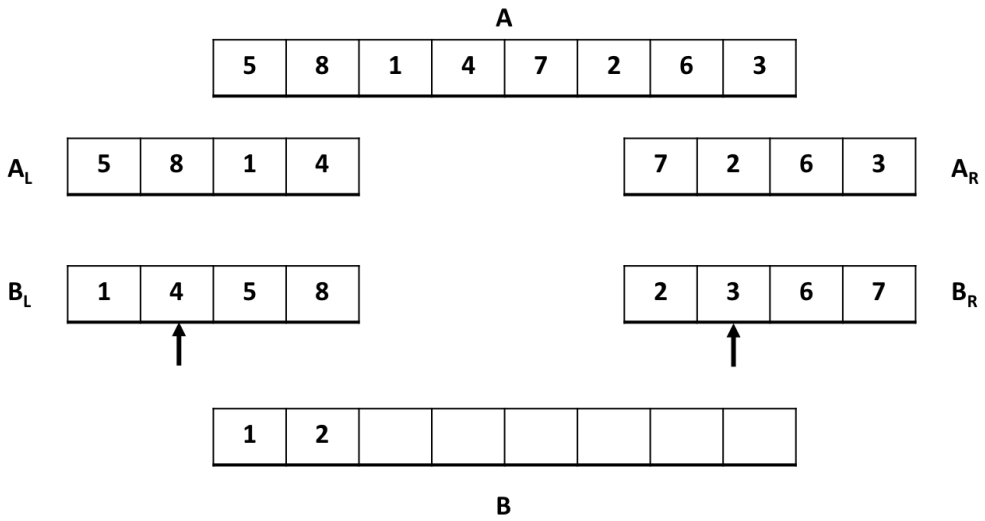
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

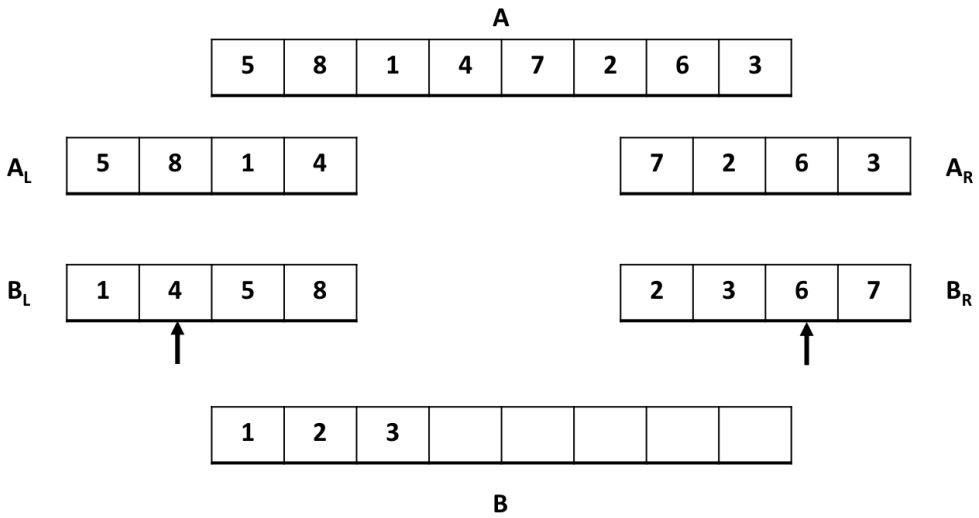
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

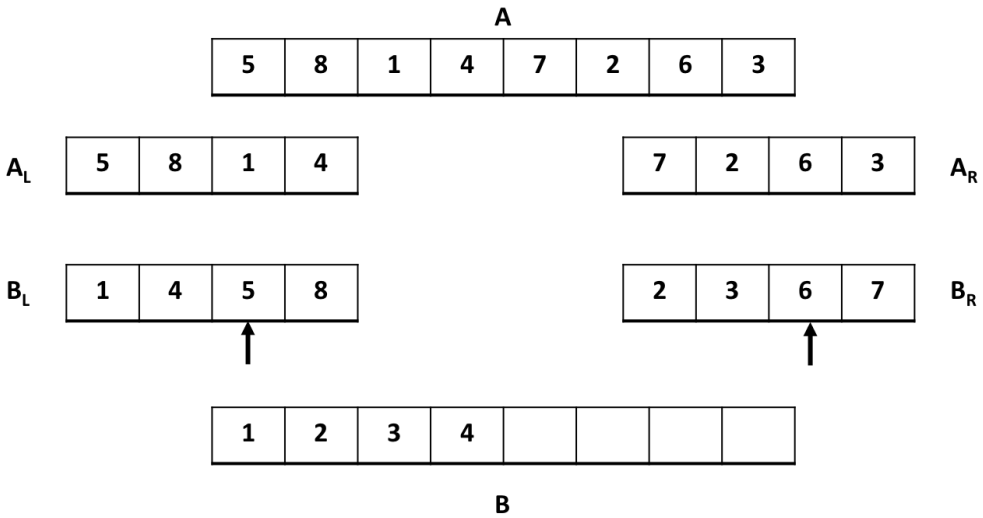
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

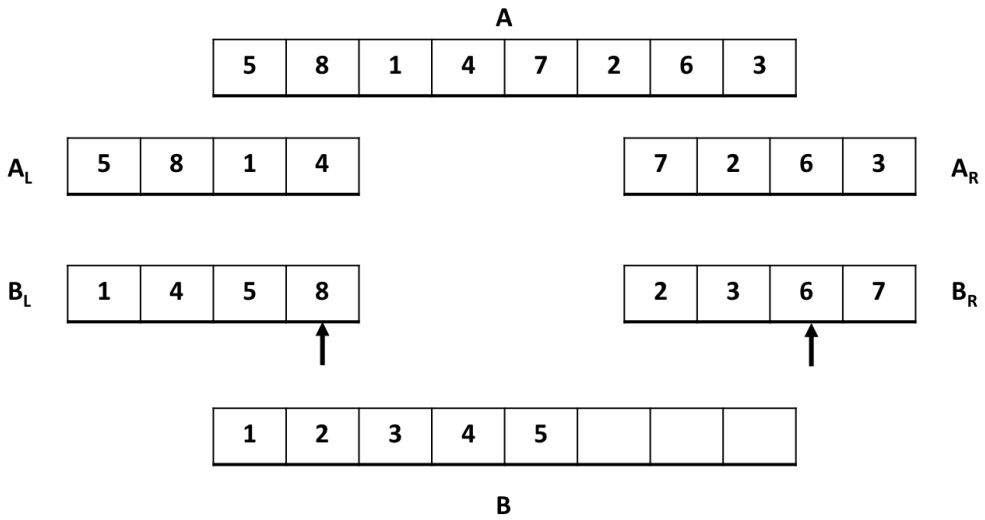
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

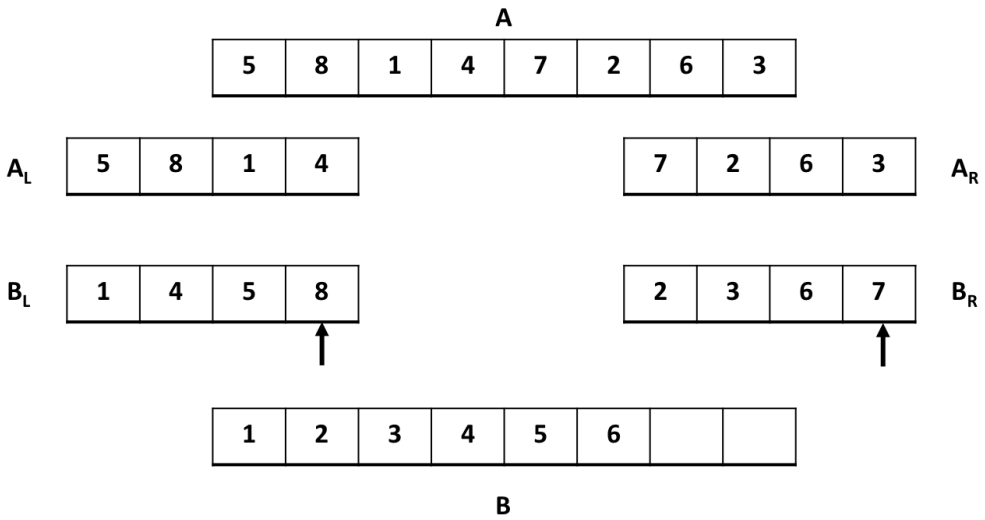
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

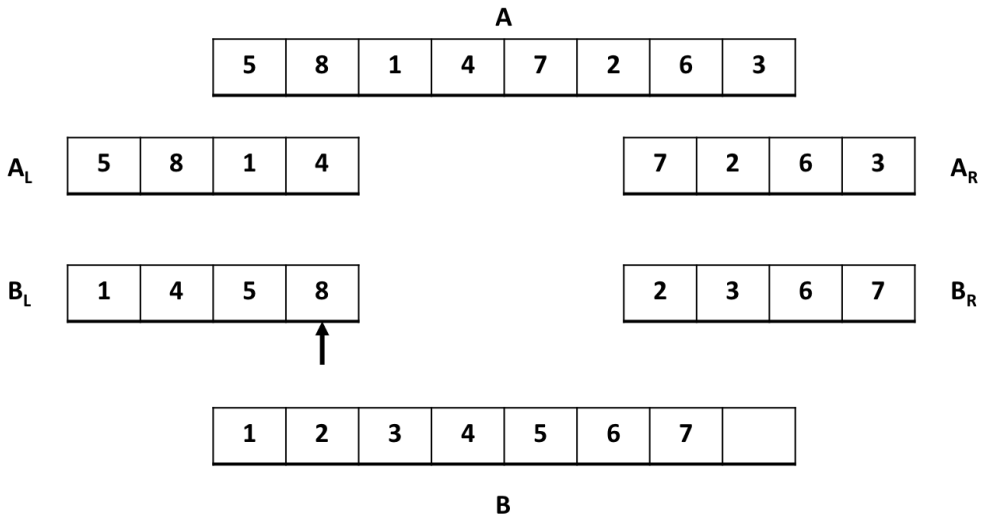
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

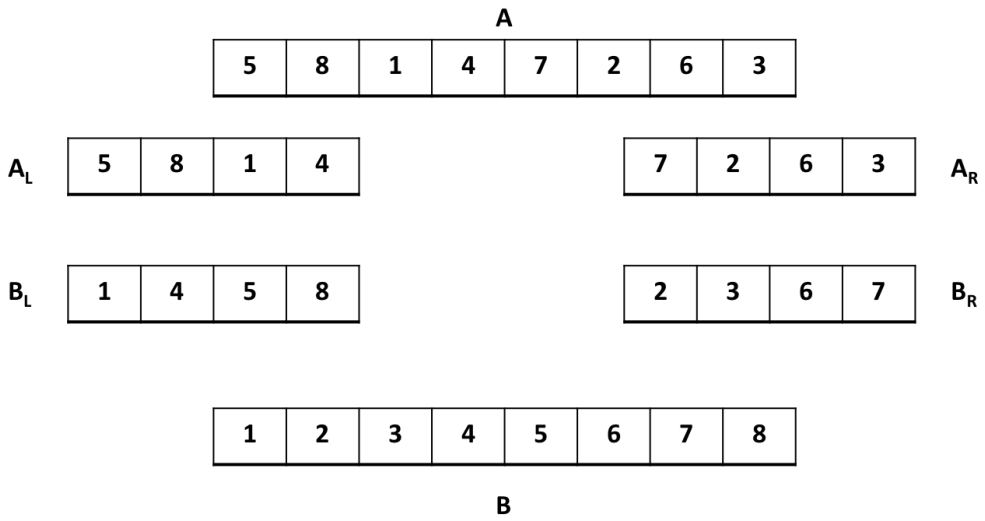
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)



Algorithm

`MergeSort`(A)

- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- How do we argue correctness?

Algorithm

`MergeSort`(A)
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- How do we argue correctness?
- Proof of correctness of Divide and Conquer algorithms are usually by induction.
  - Base case: This corresponds to the base cases of the algorithm. For the `MergeSort`, the base case is that the algorithm correctly sorts arrays of size 1.
  - Inductive step: In general, this corresponds to correctly combining the solutions of smaller subproblems. For `MergeSort`, this is just proving that the `Merge` routine works correctly. This may again be done using induction and is left as an exercise.

Algorithm

`MergeSort`(A)
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- Let n be a power of 2 (e.g., $n = 256$)
- Let $T(n)$ denote the worst case running time for the algorithm.
- Claim 1: $T(1) \leq c$ for some constant c.

Algorithm

`MergeSort`(A)
- If $(|A| = 1)$ return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort`(A_L)
- $B_R \leftarrow$ `MergeSort`(A_R)
- $B \leftarrow$ `Merge`(B_L, B_R)
- return(B)

- Let n be a power of 2 (e.g., $n = 256$)
- Let $T(n)$ denote the worst case running time for the algorithm.
- Claim 1: $T(1) \leq c$ for some constant c.
- Claim 2: $T(n) \leq 2 \cdot T(n/2) + cn$ for all $n \geq 2$.

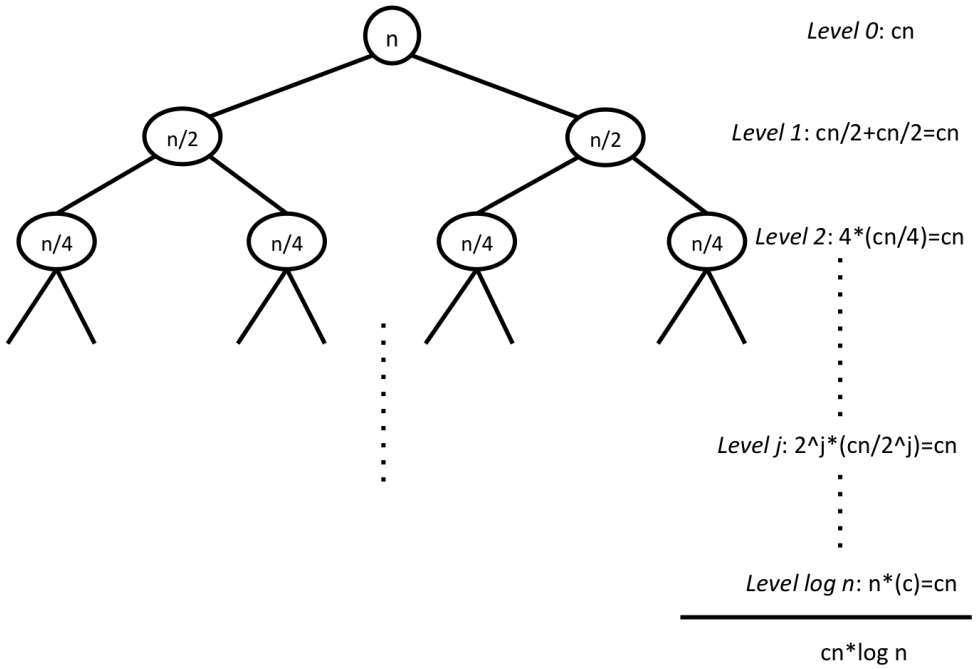
Algorithm

`MergeSort(`A`)`
- If ($|A| = 1$) return(A)
- Divide A into two equal parts A_L and A_R
- $B_L \leftarrow$ `MergeSort(`A_L`)`
- $B_R \leftarrow$ `MergeSort(`A_R`)`
- $B \leftarrow$ `Merge(`B_L, B_R`)`
- return(B)

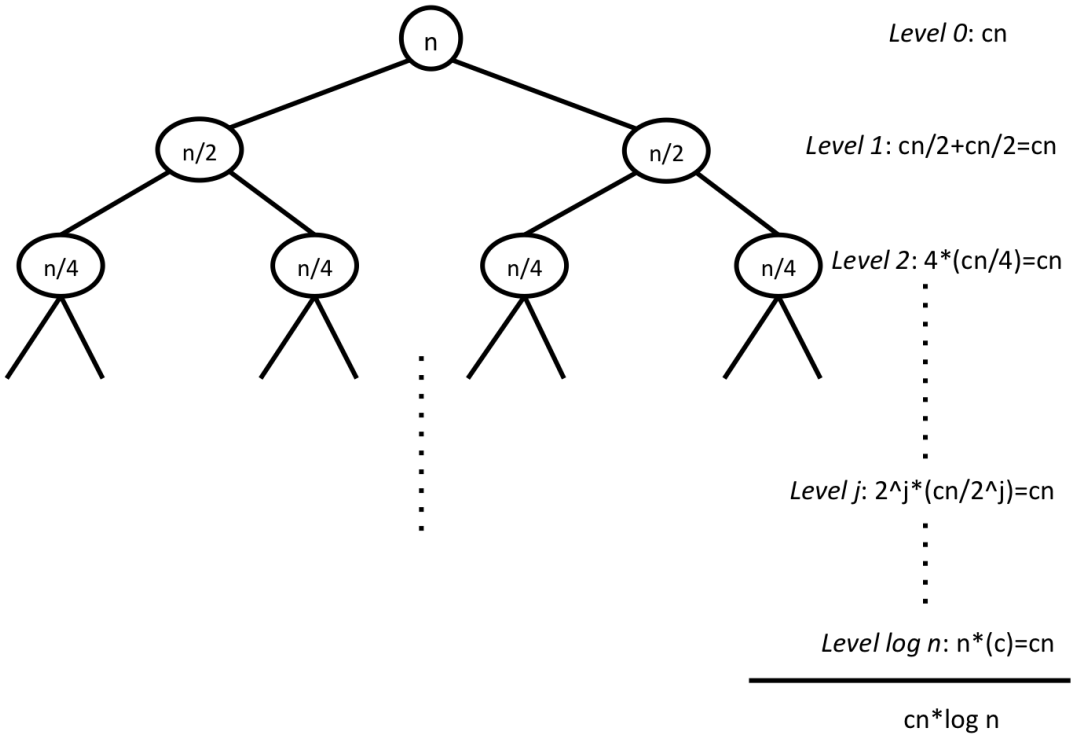
- Let n be a power of 2 (e.g., $n = 256$)
- Let $T(n)$ denote the worst case running time for the algorithm.
- Claim 1: $T(1) \leq c$ for some constant c.
- Claim 2: $T(n) \leq 2 \cdot T(n/2) + cn$ for all $n \geq 2$.
- $T(n) \leq 2 \cdot T(n/2) + cn$ for $n \geq 2$ and $T(1) \leq c$ is called a *recurrence relation* for the running time $T(n)$.
- How do we solve such recurrence relation to obtain the value of $T(n)$ as a function of n?

- Let n be a power of 2 (e.g., $n = 256$)
- Let $T(n)$ denote the worst case running time for the algorithm.
- Claim 1: $T(1) \leq c$ for some constant c.
- Claim 2: $T(n) \leq 2 \cdot T(n/2) + cn$ for all $n \geq 2$.
- $T(n) \leq 2 \cdot T(n/2) + cn$ for $n \geq 2$ and $T(1) \leq c$ is called a *recurrence relation* for the running time $T(n)$.
- How do we solve such recurrence relation to obtain the value of $T(n)$ as a function of n?
  - Unrolling the recursion: Rewrite $T(n/2)$ in terms of $T(n/4)$ and so on until a pattern for the running time with respect to all levels of the recursion is observed. Then, combine these and get the value of $T(n)$.

- Recurrence relation for Merge Sort: $T(n) \leq 2 \cdot T(n/2) + cn$ for $n \geq 2$ and $T(1) \leq c$.
- How do we solve such recurrence relation to obtain the value of $T(n)$ as a function of n?
  - Unrolling the recursion: Rewrite $T(n/2)$ in terms of $T(n/4)$ and so on until a pattern for the running time with respect to all levels of the recursion is observed. Then, combine these and get the value of $T(n)$.



- Recurrence relation for Merge Sort: $T(n) \leq 2 \cdot T(n/2) + cn$ for $n \geq 2$ and $T(1) \leq c$.
- How do we solve such recurrence relation to obtain the value of $T(n)$ as a function of n?
- So, the running time $T(n) \leq cn \cdot \log n = O(n \log n)$.



- Question: Suppose there is a divide and conquer algorithm where the recurrence relation for running time $T(n)$ is the following:

$$T(n) \leq 2T(n/2) + cn^2 \text{ for } n \geq 2, \text{ and } T(1) \leq c.$$

What is the solution of this recurrence relation in big-oh notation?

Theorem

Master Theorem: Let

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + c \cdot n^k \quad \text{and} \quad T(1) \leq c,$$

Then

$$T(n) = \begin{cases} ? & \text{if } a < b^k \\ ? & \text{if } a = b^k \\ ? & \text{if } a > b^k \end{cases}$$

End