# Basic Algorithms

**CSCI-UA 202**

Yao Xiao

Fall 2022

## Contents

# 1 9/7 Lecture

## 1.1 Insertion Sort

### 1.1.1 Overview

Input: array $A[1, \ldots, n]$ of $n$ numbers.
Goal: sort in increasing order.
For $j = 2, \cdots, n$, insert $A[j]$ into its correct position in the subarray $A[1, \ldots, j]$ by moving all elements bigger than it one position to the right.

### 1.1.2 Pseudocode

**Algorithm 1** InsertionSort($A[1, \ldots, n]$)

```
for j = 2 to n do
    key ← A[j];
    i ← j − 1;
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i];
        i ← i − 1;
    end while
    A[i + 1] ← key;
end for
```

### 1.1.3 Correctness

**Proposition 1.1.** The algorithm InsertionSort is correct.

*Proof.* We prove by induction.

- **Loop invariant.** At the beginning of the $j$th iteration (or, equivalently, end of $(j-1)$th iteration), $A[1, \ldots, j-1]$ contains the same elements that were initially there, but in sorted order.
- **Termination.** If loop invariant holds at the end of the algorithm ($j = n + 1$), then we are happy: the output is correct (sorted array).
- **Claim.** Loop invariant holds for $j = 2, 3, 4, \cdots, n + 1$.
- **Induction.** First, base of induction (initialization) when $j = 2$, $A[1]$ is sorted because there is only one element. Next, inductive step (maintenance), assume it hold for $j$. Let us prove it for $j + 1$. By assumption, $A[i, \cdots, j - 1]$ is sorted in the beginning of the $j$th iteration. During the $j$th iteration, we insert $A[j]$ into its correct position. Therefore, the loop invariant holds at the end of the $j$th iteration. □

### 1.1.4 Runtime

The worst case runtime is

$$\sum_{j=2}^{n}(c + c'j) = c(n-1) + c'\left(\frac{n(n+1)}{2} - 1\right)\frac{c'}{2}n^2 + \left(c + \frac{c'}{2}\right)n - (c + c') = \Theta(n^2).$$

# 2 9/12 Lecture

## 2.1 Big-O Notation

**Definition 2.1.** For functions $f, g : \mathbb{N} \to \mathbb{R}$, we say that

(1) $f = O(g)$ if $\exists c$, s.t. for all $n$ large enough,

$$\frac{f(n)}{g(n)} \leq c.$$

(2) $f = \Omega(g)$ if $\exists c$, s.t. for all $n$ large enough,

$$\frac{f(n)}{g(n)} \geq c.$$

(3) $f = \Theta(g)$ if $\exists c_1, c_2$, s.t. for all $n$ large enough,

$$c_1 \leq \frac{f(n)}{g(n)} \leq c_2.$$

## 2.2 Merge Sort

[Von Neumann 1945]

### 2.2.1 Pseudocode

**Algorithm 2** MergeSort($A[1, \ldots, n]$)

```
1: MergeSort(A[1, ..., n/2]);
2: MergeSort(A[n/2 + 1, ..., n]);
3: Merge both outputs;                                   ▷ Algorithm 3
```

**Algorithm 3** Merge($A[1, \ldots, m], B[1, \ldots, n], C[1, \ldots, m + n]$)

```
   /* A and B are input sorted arrays, and C is the output array */
 1: i ← 1;
 2: j ← 1;
 3: for k = 1 to m + n do
 4:     if A[i] ≤ B[j] then
 5:         C[k] ← A[i];
 6:         i ← i + 1;
 7:     else
 8:         C[k] ← B[j];
 9:         j ← j + 1;
10:     end if
11: end for
```

### 2.2.2 Correctness

**Proposition 2.2.** The algorithm Merge is correct.

*Proof.* We prove by induction.

- **Loop invariant.** At the beginning of the $k$th iteration, $C[1, \ldots, k-1]$ contains the $k-1$ smallest elements of $A \cup B$ in sorted order and these elements are $A[1, \ldots, i-1]$ and $B[1, \ldots, j-1]$.
- **Initialization.** Initially, $k = 1$, $i = j = 1$, so invariant holds trivially.
- **Maintenance.** Assume invariant hold for the $k$th iteration. We want to show that it also holds for the $(k+1)$th iteration. Assume $A[i] \leq B[j]$. Then $A[i]$ is the smallest among all remaining items. Therefore, invariant also holds at the $(k+1)$th iteration. If $A[i] > B[j]$, the case is similar.
- **Termination.** When $k = m + n + 1$ (end of the last iteration), $C$ contains all elements of $A \cup B$ in sorted order, as desired. □

**Proposition 2.3.** The algorithm MergeSort is correct.

*Proof.* The correctness follows from the correctness of Merge. □

### 2.2.3 Runtime.

**Proposition 2.4.** The algorithm Merge runs $\Theta(m + n)$ time on inputs of size $n$ and $m$ respectively.

*Proof.* This is trivial. □

**Proposition 2.5.** The algorithm MergeSort runs $\Theta(n \log n)$ time on an input of size $n$.

*Proof.* Denote by $T(n)$ the runtime on an input of size $n$ and we obtain

$$T(n) = 2T\left(\frac{n}{2}\right) + n.$$

There are three methods to solve such a recurrence relation: recursion tree, substitution method (induction), and the master theorem. Here, by any one of these methods we can obtain $T(n) = \Theta(n \log n)$ and we omit the details. □

# 3 9/14 Lecture

## 3.1 Substitution Method

Take the recurrence relation of MergeSort algorithm as an example.

**Claim.** $\forall n \geq 2$, $T(n) \leq 10n \log n$.

*Proof.* We prove by induction. The base case $n = 2$ is trivial. Assume it holds up to $n-1$, then for $n$, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + n \leq 2\left(10 \cdot \frac{n}{2} \log \frac{n}{2}\right) + n = 10n \log \frac{n}{2} + n = 10n \log n - 10n + n \leq 10n \log n.$$

□

## 3.2 The Master Theorem

**Theorem 3.1** (The master theorem)**.** For a recurrence relation $T(n) = aT(n/b) + f(n)$,

(1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(2) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

(3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$.

## 3.3 Quick Sort

- Also based on divide-and-conquer;
- "MergeSort in reverse."

### 3.3.1 Overview

(1) Select a pivot element. For us, choose the last element.

(2) Partition: everything smaller than the pivot goes left, and everything larger than the pivot goes right. Pivot stays in between.

(3) Recursively sort both parts.

### 3.3.2 Pseudocode

**Algorithm 4** QuickSort($A[1, \ldots, n]$)

```
1: k ← Partition(A);                ▷ k is the final and correct position of the pivot, Algorithm 5
2: QuickSort(A[1, ..., k − 1]);
3: QuickSort(A[k + 1, ..., n]);
```

**Algorithm 5** Partition($A[1, \ldots, n]$)

```
1: pivot ← A[n];
2: i ← 1;
3: for j = 1 to n − 1 do
4:     if A[j] ≤ pivot then
5:         swap A[i] with A[j];
6:         i ← i + 1;
7:     end if
8: end for
9: swap A[i] with A[n];
10: return i;
```

### 3.3.3 Correctness

**Proposition 3.2.** The algorithm PARTITION is correct.

*Proof.* We prove by induction.

- **Loop invariant.** At the beginning of the $j$th iteration, for all $k \in \{1, \cdots, i-1\}$, $A[k] \leq$ pivot, and for all $k \in \{i, \cdots, j-1\}$, $A[k] >$ pivot.
- **Exercise.** The correctness of PARTITION is left for readers to prove using this loop invariant by induction. □

**Proposition 3.3.** The algorithm QUICKSORT is correct.

*Proof.* The correctness follows from the correctness of PARTITION. □

**Remark 3.4.** (1) QUICKSORT is in place, since no extra memory is used.

(2) QUICKSORT is not stable, at least no with this in-place implementation.

### 3.3.4 Runtime

**Proposition 3.5.** The algorithm PARTITION runs $\Theta(n)$ time on an input of size $n$.

*Proof.* This is trivial. □

**Proposition 3.6.** On an input of size $n$, the algorithm QUICKSORT runs $\Theta(n \log n)$ time best case and runs $\Theta(n^2)$ time worst case.

*Proof.* In the worst case, the array is already sorted. In this case, the partition is highly unbalanced, and the recurrence for runtime is

$$T(n) = T(n-1) + T(0) + n \implies T(n) = \Theta(n^2).$$

In the best case, the pivot is the median, in which case the recurrence is

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \implies T(n) = \Theta(n \log n).$$

□

# 4 9/19 Lecture

## 4.1 Quick Sort Runtime Analysis

The worst case when the array is sorted runs $\Theta(n^2)$ time and the best case when the pivot is the median in every iteration runs $\Theta(n \log n)$ time. Now we investigate some average case for a shuffled array and assume all elements are distinct. In this case, the last element in the array is unlikely to be in the top or bottom 10% percentiles. So with probability 80%, the last element is not there. For simplicity, assume that the pivot is exactly in 10th percentile. Then

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn.$$

From here we can deduce that the total runtime is $T(n) = O(n \log_{10/9} n)$ and $T(n) = \Omega(n \log_{10} n)$. Hence the runtime of QUICKSORT is $\Theta(n \log n)$ with high probability.

**Remark 4.1.** (1) In practice, people use a heuristic that uses the median of {first, center, last} elements. (This works well in many cases, including sorted arrays.)

(2) Choose pivot at random. The expected runtime is $\Theta(n \log n)$ in the worst case no matter what the input is.

(3) Alternatively, choose the median of the whole array as the pivot. The runtime is $\Theta(n \log n)$, and it turns out that you can find the median without sorting in $\Theta(n)$ time.

## 4.2 Integer Multiplication

[Karatsuba]

### 4.2.1 Overview

The bad algorithm to compute $a \times b$ is to add $a$ for $b$ times, so that the runtime is $O(b) = O(10^n)$, where $n$ is the input size (the number of digits). Now let us use divide and conquer. To multiply $a$ and $b$ where $b$ has $n$ digits, we split them into high digits and low digits such that

$$a = a_n \cdot 10^{n/2} + a_1,$$
$$b = b_n \cdot 10^{n/2} + b_1.$$

Then,

$$a \cdot b = a_n b_n \cdot 10^n + (a_n b_1 + a_1 b_n) \cdot 10^{n/2} + a_1 b_1.$$

### 4.2.2 Pseudocode

**Algorithm 6** MULT$(a, b)$

1: $H \leftarrow$ MULT$(a_n, b_n)$;
2: $M \leftarrow$ MULT$(a_n + a_1, b_n + b_1)$;
3: $L \leftarrow$ MULT$(a_1, b_1)$;
4: **return** $H \cdot 10^n + (M - L - H) \cdot 10^{n/2} + L$;

### 4.2.3 Correctness

**Proposition 4.2.** The algorithm MULT is correct.

*Proof.* The correctness is already verified in the overview section. □

### 4.2.4 Runtime

**Proposition 4.3.** The algorithm MULT runs $\Theta(n^2)$ time on an input of size $n$.

*Proof.* The recursion is $T(n) = 3T(n/2) + \Theta(n)$, because there are 3 recursive calls on MULT and 6 other addtions and subtractions. This solves to $T(n) = \Theta(n^{\log_2 3})$ by case (1) of the master theorem. □

# 5 9/21 Lecture

## 5.1 Closest Pair of Points

### 5.1.1 Overview

**Problem 5.1** (one-dimensional)**.** Given $n$ numbers $p_1, p_2, \cdots, p_n \in \mathbb{Z}$, find $i$, $j$, $i \neq j$, such that

$$\text{dist}(p_i, p_j) = |p_i - p_j|$$

is as small as possible.

A naive algorithm is to try all pairs $i < j$ and output the closest pair. The runtime would be $\binom{n}{2} = \Theta(n^2)$. A better algorithm is to sort the numbers and then check for the closest pair among consecutive numbers. The correctness holds because the closest pair must be consecutive after sorting. The runtime is $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$.

**Remark 5.2.** This can actually be solved in $\Theta(n)$ time on an input of size $n$.

**Problem 5.3** (two-dimensional)**.** Given $n$ points $p_1, p_2, \cdots, p_n \in \mathbb{Z}^2$, find a pair of points that are the closest in Euclidean distance

$$\text{dist}(p_i, p_j) = \sqrt{((p_i)_x - (p_j)_x)^2 + ((p_i)_y - (p_j)_y)^2}.$$

A naive algorithm is as before. We try all pairs and the runtime is $\Theta(n^2)$. Next we propose a better algorithm.

**Remark 5.4.** For simplicity, we assume that no two points share the same $x$-coordinate.

(1) Divide points into two halves using a vertical line $L$ (i.e., based on $x$ coordinate). This can be done using a sorting algorithm ($O(n \log n)$) or a median finding algorithm ($O(n)$).

(2) Conquer: Recursively find the closest pair in the left half and in the right half. Let $\delta_1$ and $\delta_2$ be the distances found. Define $\delta = \min\{\delta_1, \delta_2\}$.

(3) Combine: Only care about points in a strip around $L$. That is like a 1-dimensional problem.

  (a) Take all points in a strip of width $2\delta$ around $L$. There are $\Theta(n)$ of them.

  (b) Sort them by $y$ coordinate. Let $q_1, \cdots, q_k$ be the sorted points. This is $\Theta(n \log n)$ time.

  (c) For each point $q_i$, compute its distance $q_{i+1}, \cdots, q_{i+11}$. This is $\Theta(n)$ time.

  (d) Let $\delta_{1,2}$ be the shortest distance found.

(4) Return $\min\{\delta, \delta_{1,2}\}$.

### 5.1.2 Correctness

**Proposition 5.5.** In (3c), if $j > i + 11$, then $\text{dist}(q_i, q_j) \geq \delta$.

*Proof.* We partition the strip into squares of side length $\delta/2$. Each square contains at most one point, because the diameter of the square is $\delta/\sqrt{2} < \delta$. Also, points that are 3 or more rows below $q_i$ must be at distance $\geq \delta$ from $q_i$, so we do not care about them. This leaves at most 11 points that appear in the sorted order after $q_i$ (4 squares each row, at most 3 rows, and excluding the square of $q_i$ itself). □

**Proposition 5.6.** The procedure provided above for solving the closest pair problem in two dimensions is correct.

*Proof.* The correctness is trivial given Proposition 5.5. □

### 5.1.3 Runtime

**Proposition 5.7.** The runtime of the procedure provided above for solving the closest pair problem in two dimensions is $\Theta(n \log^2 n)$ on an input of size $n$.

*Proof.* We have the recurrence relation from the discussion above that $T(n) = 2T(n/2) + \Theta(n \log n)$, where $\Theta(n \log n)$ is the time for sorting in (3b). This recurrence does not fit the master's theorem, but can be easily solved by recursion tree that $T(n) = \Theta(n \log^2 n)$. □

**Remark 5.8.** Note that $\Theta(n \log^2 n)$ is not optimal for solving this problem. We can avoid sorting in each recursive call by sorting once for all by $y$ coordinate in the beginning, taking $\Theta(n \log n)$ time. Then the recurrence becomes $T(n) = 2T(n/2) + \Theta(n)$, which by the master's theorem, solves to $T(n) = n \log n$.

# 6 9/26 and 9/28 Lectures

## 6.1 Median and Selection

### 6.1.1 Overview

Input: a set of $n$ distinct numbers and an index $i \in \{1, \cdots, n\}$.
Goal: output the $i$th smallest element.

- The case when $i = n/2$ is the median.
- If $i = 1$ or $i = n$, then the problem can be easily solved in linear time.
- A linear time (deterministic) algorithm would lead to $\Theta(n \log n)$ time (deterministic) quick sort algorithm.

### 6.1.2 Pseudocode

**Algorithm 7** RandSelect$(A, i)$

```
1: j ← ApproxMedian(A);                                   ▷ Algorithm 8
2: k ← Partition(A, j);                                   ▷ Algorithm 5
3: if k = i then
4:     return A[k];
5: else if k > i then
6:     return RandSelect(A[1, ..., k − 1], i);
7: else
8:     return RandSelect(A[k + 1, ..., n], i − k);
9: end if
```

[Blum et al. 1973]

**Algorithm 8** ApproxMedian$(A)$

```
1: Partition A into groups of size 5;
2: Find median in each group to form an array A′ of n/5 medians;
3: return ApproxMedian(A′);
```

**Algorithm 9** Select$(A, i)$

```
1: Partition A into groups of size 5;
2: Find median in each group to form an array A′ of n/5 medians;
3: x ← Select(A′, n/10);
4: k ← rank_A(x);                          ▷ the rank of x in A, i.e., x = A[k]
5: if k = i then
6:     return x;
7: else if k > i then
8:     return Select(A[1, ..., k − 1], i);
9: else
10:    return Select(A[k + 1, ..., n], i − k);
11: end if
```

### 6.1.3 Correctness

**Proposition 6.1.** The algorithm RandSelect is correct.

*Proof.* Note that the correctness is trivial, and RandSelect is guaranteed to output the $i$th smallest element no matter how ApproxMedian is implemented. □

**Proposition 6.2.** The algorithm ApproxMedian is correct, i.e., ApproxMedian returns an element that is not in the top or bottom 30 percentiles.

*Proof.* We can see that after arrangement (Figure 1), the output has $\frac{n}{10}$ medians larger than it, and each one has two more elements from its group that are even bigger. Therefore, $\frac{3n}{10}$ elements are larger than the output, and similarly $\frac{3n}{10}$ are smaller. □

**Proposition 6.3.** The algorithm Select is correct.

*Proof.* Select is a combination of RandSelect and ApproxMedian, and the correctness directly follows. □

### 6.1.4 Runtime

**Proposition 6.4.** The algorithm Select runs $\Theta(n)$ time on an input of size $n$.

*Proof.* The call of ApproxMedian on an input of size $n$ takes $\Theta(n)$ time. Partition also takes $\Theta(n)$ time. □

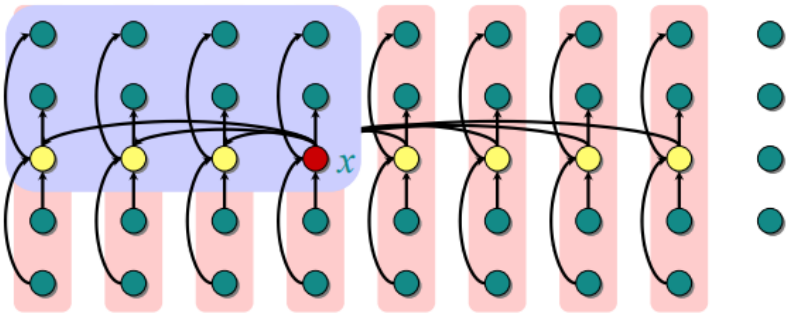



Figure 1: Schema of the algorithm ApproxMedian

**Proposition 6.5.** The algorithm ApproxMedian runs $\Theta(n)$ time on an input of size $n$.

*Proof.* The subproblem size is 1/5 the original problem size, with only one subproblem each recursive step. The non-recursive cost on an input of size $n$ is linear since we only need to find the median of 5 elements for $n/5$ groups respectively. Therefore, we have the recurrence relation $T(n) = T(n/5) + \Theta(n)$, which by master's theorem, solves to $T(n) = \Theta(n)$. □

**Proposition 6.6.** The algorithm Select runs $\Theta(n)$ time on an input of size $n$.

*Proof.* Finding the medians of the $\frac{n}{5}$ groups takes $\Theta(n)$ time, since it takes constant time to find the median of constant number of values. Another $\Theta(n)$ time is needed to traverse $A$ to find the rank of $x$. The recursive calls have size $\frac{n}{5}$ and $\frac{7n}{10}$, respectively, where the latter follows from the proof of correctness of ApproxMedian. Therefore, we have the recurrence relation $T(n) = T(n/5) + T(7n/10) + \Theta(n)$, which solves to $T(n) = \Theta(n)$ by recursion tree or substitution method. □

# 7 10/3 Lecture

## 7.1 Finding the Largest Element

### 7.1.1 Overview

An algorithm to find the largest element with $(n-1)$ comparison queries is easy to come up with. Is it possible to use fewer comparison queries? Less than $n/2$ is clearly impossible since we need to ask about all items at least once.

### 7.1.2 Lower Bound

**Proposition 7.1.** Computing the maximum of $n$ elements requires at least $(n-1)$ comparison queries.

*Proof.* On input $1, \cdots, n$ the algorithm must output the $n$th element. During the algorithm, for each $i = 1, \cdots, n-1$, there must be a comparison of the $i$th element to an element to the right of it. In other words, it must "lose" at least once. Otherwise, we can imagine modifying the $i$th element to contain $(n+1)$, in which case the algorithm sees no difference and would therefore output the wrong answer. In conclusion, the total number of comparison queries must be at least $(n-1)$. □

## 7.2 Decision Tree

Every comparison-based algorithm can be written as a decision tree. The depth is the worst case number of comparison queries that the algorithm performs. For example, Figure 2 shows the decision tree for sorting 3 elements (or finding the ranking).

**Theorem 7.2.** Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparison queries.

**Remark 7.3.** This shows that MergeSort is asymptotically optimal. Like InsertionSort and QuickSort, MergeSort is a comparison-based algorithm. Therefore, they all must perform $\Omega(n \log n)$ comparisons implying their running time is $\Omega(n \log n)$.

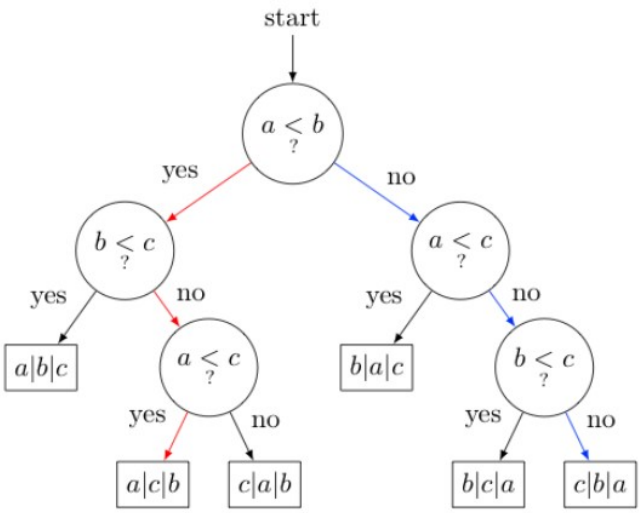



Figure 2: Decision tree for sorting 3 elements

*Proof.* When sorting $n$ elements, any correct comparison-based must have at least $n!$ leaves since there are $n!$ possible outputs, i.e., the number of permutations of $n$ elements. Also note that the number of leaves in any binary tree of depth $k$ is at most $2^k$. Therefore, if $k$ denotes the depth, then $2^k \geq n!$, which means $k \geq \log(n!) = \Theta(n \log n)$. □

# 8 10/5 Lecture

## 8.1 Counting Sort

### 8.1.1 Overview

Assume keys are in $\{0, \cdots, k-1\}$, we try to further reduce the time complexity of the sorting algorithm.

### 8.1.2 Pseudocode

**Algorithm 10** CountSort($A$)

```
1: L ← array of k empty lists;
2: for j = 0 to n − 1 do
3:     L[A[j].key].append(A[j].val);
4: end for
5: output = ∅;
6: for i = 0 to k − 1 do
7:     output.extend(L[i]);
8: end for
9: return output;
```

### 8.1.3 Correctness

**Proposition 8.1.** The algorithm CountSort is correct.

*Proof.* The correctness is obvious. We count the number of occurrences of each key, then output them in sequence. □

**Remark 8.2.** CountSort is stable with most implementations.

### 8.1.4 Runtime

**Proposition 8.3.** The algorithm CountSort runs $\Theta(n + k)$ time on an input of size $n$ with the assumption.

*Proof.* The first loop iterates $n$ times. The second loop runs $\Theta(n+k)$ time since it iterates $k$ times and in each iteration we extend a list of at most $n$ elements. Therefore, the total runtime is $T(n) = \Theta(n+k)$. Note that if $k = O(n)$, then $T(n) = \Theta(n)$. □

## 8.2 Radix Sort

### 8.2.1 Overview

CountSort is bad when $k = \Omega(n^2)$. Assume now that keys are in $\{0, 1, \cdots, k^d - 1\}$ for some numbers $k, d > 0$. We can think of them as $d$-digit numbers where each digit is in $\{0, \cdots, k-1\}$.

### 8.2.2 Pseudocode

**Algorithm 11** RadixSort($A$)

```
for i = 0 to d − 1 do
    stable sort A based on the ith digit;
end for
```

### 8.2.3 Correctness

As an example, see Figure 3.

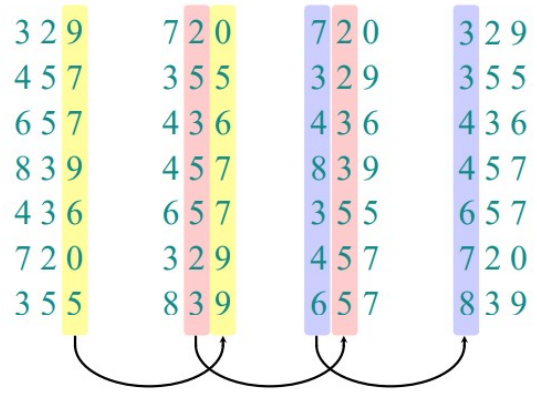



Figure 3: An example of the procedure of RadixSort

**Proposition 8.4.** The algorithm RadixSort is correct.

*Proof.* The correctness of RadixSort follows by induction on the column being sorted, and that is why we require the sorting of each digit to be stable. □

### 8.2.4 Runtime

**Proposition 8.5.** The algorithm RadixSort runs $\Theta(d(n+k))$ time on an input of size $n$ with the assumption.

*Proof.* The loop iterates for $d$ times, and in each iteration, we use CountSort (Algorithm 10) which runs $\Theta(n+k)$ times. Therefore, the total runtime of RadixSort is $T(n) = \Theta(d(n+k))$. Note that if $d$ is constant and $k = O(n)$, then $T(n) = \Theta(n)$. □

## 8.3 Fibonacci Numbers

### 8.3.1 Overview

Fibonacci numbers $F_n$ satisfy $F_1 = F_2 = 1$, and for all $n \geq 3$, $F_n = F_{n-1} + F_{n-2}$. A naive algorithm is to directly apply the recursion of the Fibonacci numbers, as is shown in Algorithm 12.

**Algorithm 12** SlowFib($n$)

```
if n ≤ 2 then
    return 1;
else
    return SlowFib(n − 1) + SlowFib(n − 2);
end if
```

However, SlowFib has exponential runtime, since subproblems are repetitively computed.

### 8.3.2 Dynamic Programming

The first version is a memorized version of the Fibonacci algorithm, as is shown in Algorithm 13. With a dictionary memo (initially empty) to cache the computed subproblems, we can avoid the massive recomputations. The runtime complexity is $T(n) = \Theta(n)$, since there are $n$ subproblems and each requires constant time.

**Algorithm 13** TopDownFib($n$)

```
if n ∈ memo then
    return memo[n];
end if
if n ≤ 2 then
    return 1;
else
    return TopDownFib(n − 1) + TopDownFib(n − 2);
end if
```

There is an even simpler bottom-up implementation of such dynamic programming as is shown in Algorithm 14.

**Algorithm 14** BottomUpFib($n$)

```
fibs ← [];
for k = 1 to n do
    if k ≤ 2 then
        fibs[k] = 1;
    else
        fibs[k] = fibs[k − 1] + fibs[k − 2];
    end if
end for
return fibs[n];
```

Here we used $\Theta(n)$ memory. This can actually be improved to constant space by keeping track of only the previous two terms instead of the whole sequence.

# 9 10/11 Lecture

## 9.1 Rod Cutting

### 9.1.1 Overview

Input: the array of prices for each length $p_1, \cdots, p_n$.
Goal: compute the optimal revenue $r_n$ obtainable from a rod of $n$ feet.
The number of partitions of a rod of length $n$ is $2^{n-1}$, and even if we ignore duplicate partitions (differing in permutation), we still have

$$\frac{e^{\pi\sqrt{2n/3}}}{4\sqrt{3}n} = \Theta(e^{\sqrt{n}}) \quad \text{possible partitions [Hardy Ramanujan 1918].}$$

**Remark 9.1.** A natural **greedy** approach is to take the piece with highest price per foot in each step. However, this may not give the optimal solution.

### 9.1.2 Dynamic Programming

**Subproblems.** $r_0, r_1, \cdots, r_n$, where $r_j$ is the optimal revenue for a rod of length $j$.

**Recurrence.** We start with stating an observation on the optimal substructure. If the optimal revenue $r_n$ is achieved by a partition whose first piece is of length $i$, then

$$r_n = p_i + r_{n-i}.$$

Hence we have the recurrence

$$r_n = \begin{cases} 0, & \text{if } n = 0, \\ \max\{p_1 + r_{n-1}, p_2 + r_{n-2}, \cdots, p_n + r_0\}, & \text{otherwise.} \end{cases}$$

**Algorithm.** Here is the top-down implementation, keeping a dictionary memo (initially empty) to cache the computed subproblems, as is shown in Algorithm 15.

**Algorithm 15** TopDownCutRod($n$)

```
1: if n ∈ memo then
2:     return memo[n];
3: end if
4: if n = 0 then
5:     return 0;
6: else
7:     q ← −∞;
8:     for i = 1 to n do
9:         q ← max{q, p_i + TopDownCutRod(n − i)};
10:    end for
11:    memo[n] ← q;
12:    return q;
13: end if
```

The simpler bottom-up implementation is shown in Algorithm 16.

**Algorithm 16** BottomUpCutRod($n$)

```
1: r ← an empty array of size (n + 1);
2: r[0] ← 0;
3: for j = 1 to n do
4:     q ← −∞;
5:     for i = 1 to j do
6:         if p_i + r[j − i] > q then
7:             q ← p_i + r[j − i];
8:         end if
9:     end for
10:    r[j] ← q;
11: end for
12: return r[n];
```

The runtime of both algorithms is $\Theta(n^2)$, which is easy to verify by counting the number of iterations.

**Remark 9.2.** In both approaches, it is easy to modify the algorithm to output the optimal partition, in addition to the optimal revenue.

**Remark 9.3.** Instead of guessing the first cut, we could guess any cut. The subproblems are the same, but the guess is different, leading to a different recurrence

$$r_n = \max\{r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1, p_n\}.$$

# 10 10/12 Lecture

## 10.1 Longest Common Subsequence

### 10.1.1 Overview

The Longest Common Subsequence problem is abbreviated as LCS. Given two strings $X_{1...m}$ and $Y_{1...n}$, find the length of the longest common subsequence. A nice demo can be found here, and examples are shown in Figure 4.

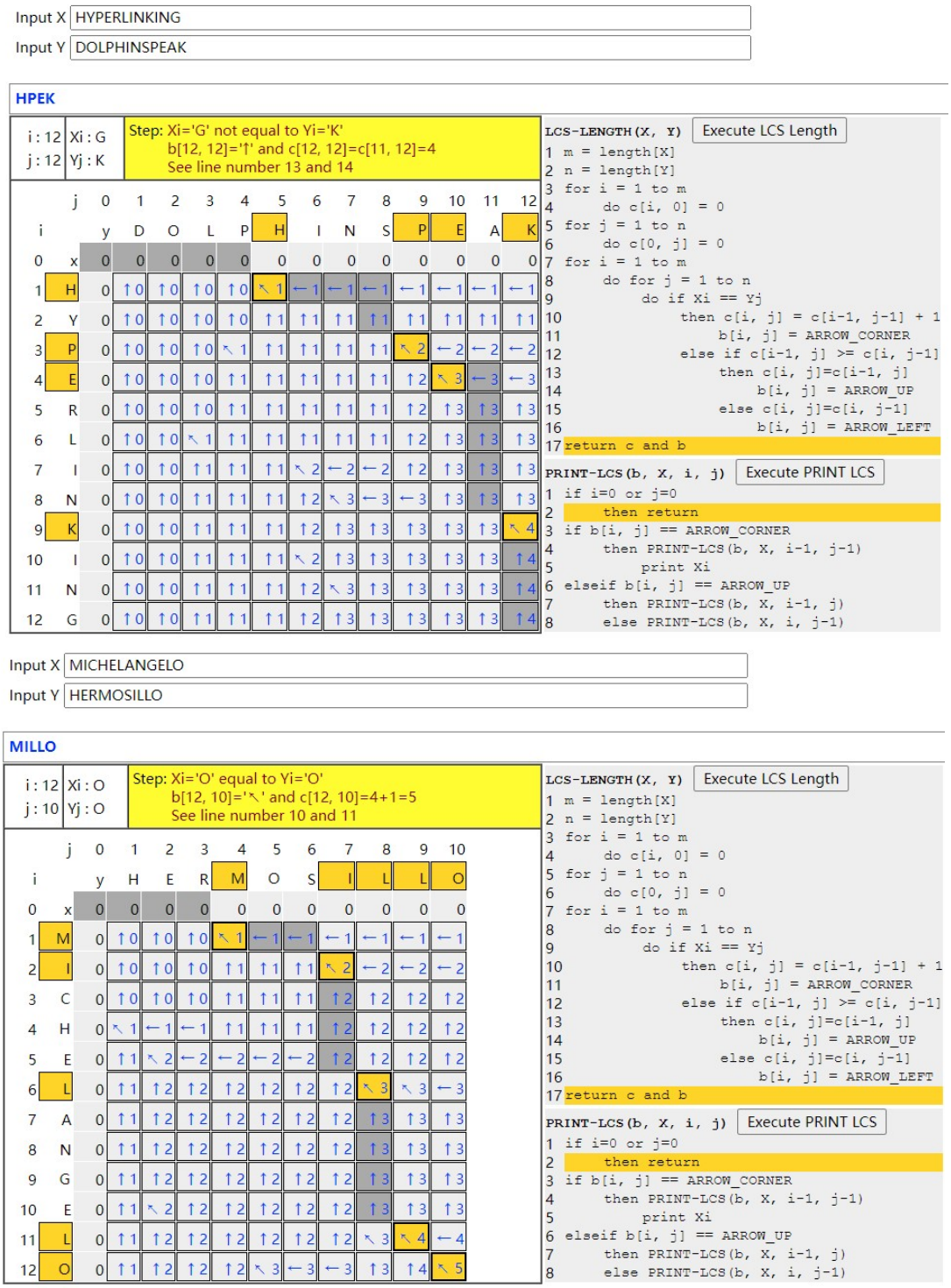



Figure 4: Examples of the output and mechanism of the LCS problem

### 10.1.2 Dynamic Programming

**Subproblems.** For $i = 0, \cdots, m$ and $j = 0, \cdots, n$, let $c[i][j]$ be the length of the longest common subsequence of $X_{1\ldots i}$ and $Y_{1\ldots j}$.

**Recurrence.** We have the recurrence

$$c[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1][j-1] + 1, & \text{if } i > 0,\ j > 0, \text{ and } X_i = Y_j, \\ \max\{c[i][j-1], c[i-1][j]\}, & \text{if } i > 0,\ j > 0, \text{ and } X_i \neq Y_j. \end{cases}$$

**Algorithm.** The bottom-up implementation is shown in Algorithm 17.

**Algorithm 17** LCS($n$)

```
r ← an empty array of size (m + 1) × (n + 1);
for i = 0 to m do
    c[i][0] ← 0;
end for
for j = 0 to n do
    c[0][j] ← 0;
end for
for i = 1 to m do
    for j = 1 to n do
        if X_i = Y_j then
            c[i][j] ← c[i − 1][j − 1] + 1;
        else
            c[i][j] ← max{c[i][j − 1], c[i − 1][j]};
        end if
    end for
end for
return c[m][n];
```

The runtime of LCS (Algorithm 17) is $\Theta(mn)$ by simple counting of the number of iterations.

**Remark 10.1.** We can retrieve the actual sequence using the arrows as is shown in Figure 4.

# 11 10/17 Lecture

## 11.1 Knapsack

### 11.1.1 Overview

Input: a list of weights $w_1, \cdots, w_n$ with a corresponding list of values $v_1, \cdots, v_n$; the weight limit $W$.
Goal: find a subset of $\{1, \cdots, n\}$ with maximum values and total weight within the weight limit $W$.
A greedy approach may be to repeatedly pick the item that maximizes the ratio $v_i/w_i$, but this may not be optimal.

### 11.1.2 Dynamic Programming

**Subproblems.** For $i = 0, 1, \cdots, n$ and $w = 0, 1, \cdots, W$, define $\mathrm{DP}[i][w]$ as the maximum value achievable with items $1, \cdots, i$ and weight limit $W$.

**Recurrence.** We have the recurrence

$$\mathrm{DP}[i][w] = \begin{cases} 0, & \text{if } i = 0, \\ \mathrm{DP}[i-1][w], & \text{if } w_i > w, \\ \max\{\mathrm{DP}[i-1][w], v_i + \mathrm{DP}[i-1][w-w_i]\}, & \text{otherwise.} \end{cases}$$

**Algorithm.** The bottom-up implementation is left as an exercise. The runtime is $\Theta(nW)$ by simply counting the number of iterations.

**Remark 11.1.** Notice that this runtime is not polynomial. The input is of size $\log W$ instead of $W$ if the value is given in binary. To be truly polynomial, we will need runtime $\Omega(n^k \log^l n)$ for some $k, l \in \mathbb{R}$. Our dynamic programming algorithm is "pseudo-polynomial." Knapsack is actually NP-complete, and we do not expect polynomial-time algorithm.

# 12 10/19 Lecture

## 12.1 Interval Scheduling

### 12.1.1 Overview

Input: a list of intervals $S = \{I_1, \cdots, I_n\}$, where $I_i = [s_i, f_i)$.
Goal: find the largest subset $S' \subseteq S$ of non-intersecting intervals.

**Dynamic Programming 1.** For each $i, j \in \{1, \cdots, n\}$, consider the subproblem $\mathrm{DP}[i][j]$ as the optimal number of intervals we can schedule among intervals starting after $f_i$ and ending before $s_j$. The recurrence relation would thus be

$$\mathrm{DP}[i][j] = \max_{k \text{ s.t. } [s_k, f_k) \subseteq [f_i, s_j)} (\mathrm{DP}[i][k] + \mathrm{DP}[k][j] + 1).$$

The top-down implementation is obvious, and the bottom-up approach may require appropriate sorting of the subproblems. The runtime is $n^2 \cdot \Theta(n) = \Theta(n^3)$.

**Dynamic Programming 2.** We sort the intervals by their finishing time. Then consider the subproblem $\mathrm{DP}[i]$ representing the largest number of intervals one can schedule among $I_1, \cdots, I_i$ (sorted). The recurrence relation would thus be

$$\mathrm{DP}[i] = \max\{\mathrm{DP}[i-1], \mathrm{DP}[k_i] + 1\},$$

where $k_i$ is the index (sorted) of the last interval to finish before $s_i$, and with base case $\mathrm{DP}[0] = 0$. The runtime of this dynamic programming approach is $\Theta(n \log n)$. Sorting the intervals by their finishing time in advance requires $\Theta(n \log n)$ time, the number of subproblems is $n$, and we need binary search to identify $k_i$ which is $\Theta(\log n)$ for each subproblem. Hence, the total runtime is $\Theta(n \log n) + n \cdot \Theta(\log n) = \Theta(n \log n)$.

### 12.1.2 Greedy Algorithm

**Greedy Approach.** It turns out that we can always take the first interval to finish.

**Proposition 12.1.** For any list of intervals $S$, there is an optimal solution that includes the first interval to finish.

*Proof.* Take any optimal solution. If it already includes that interval, then we are done. Otherwise, we use the first interval to finish instead of the first interval in that optimal solution. □

**Algorithm.** A simple implementation of the above greedy approach is shown in Algorithm 18.

**Algorithm 18** IntervalScheduling($n$)

```
Sort by finish time;
cur_finish ← −∞;
for i = 1 to n do
    if s_i > cur_finish then
        include I_i;
        cur_finish ← f_i;
    end if
end for
```

**Remark 12.2.** In the **weighted** variant of the interval scheduling problem, each interval comes with some value, and the goal is to maximize the total value. For such a problem, dynamic programming can be extended, but greedy algorithm cannot.

# 13 10/31 Lecture

## 13.1 Huffman Coding

### 13.1.1 Overview

Normally, computers use 8 bits per symbol. If we only are about digits (10) and upper and lower case English ($2 \times 26$), you can use just 6 bits since $2^6 > 62$. This is called "fixed length encoding." We can also use **variable length encoding**. For instance, let

$$e \mapsto 1, \quad a \mapsto 01, \quad b \mapsto 010.$$

Then 0101 can be $aa$ or $be$. To avoid such ambiguities, we always work with prefix codes defined as follows.

**Definition 13.1.** A prefix code for a set of symbols $S$ is a function

$$c : S \mapsto \{0,1\}^*, \quad \text{(bit strings of any length)}$$

such that for all distinct $x, y \in S$ it holds that $c(x)$ is *not* a prefix of $c(y)$.

**Remark 13.2.** The code above is not a prefix code, because $c(a)$ is a prefix of $c(b)$.

Suppose now that we have a file with these symbol frequencies:

$$f_a = 0.32, \quad f_e = 0.25, \quad f_k = 0.20, \quad f_r = 0.18, \quad f_u = 0.05.$$

With fixed length encoding, we need 3 bits to encode 5 symbols, so we get 3 bits/symbol. However, if we encode

$$a \mapsto 11, \quad e \mapsto 01, \quad k \mapsto 001, \quad r \mapsto 10, \quad u \mapsto 0000,$$

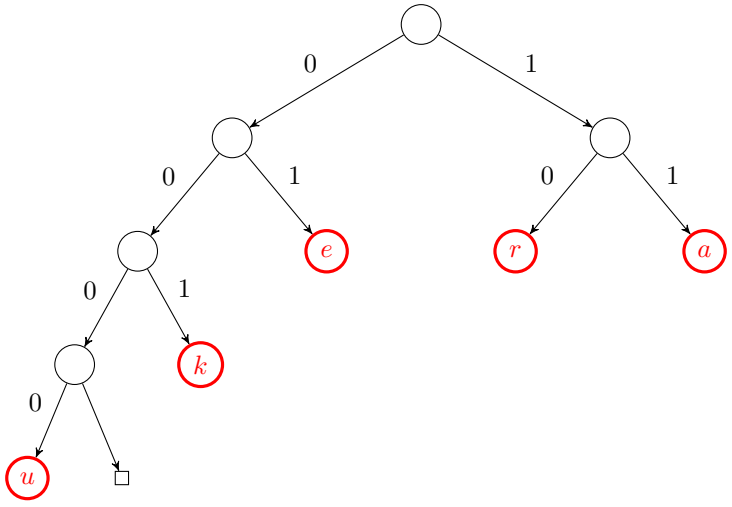
we will get $2f_a + 2f_e + 3f_k + 2f_r + 4f_u = 2.3$ bits/symbol. More generally, we give the following definition.

**Definition 13.3.** The average bits per symbol is defined as

$$\text{ABL}(c) = \sum_{x \in S} f_x \cdot |c(x)|,$$

where $c$ is the code, $S$ is the set of symbols, $f_x$ is the frequency of the symbol $x$, and $|c(x)|$ is the number of bits used to encode $x$.

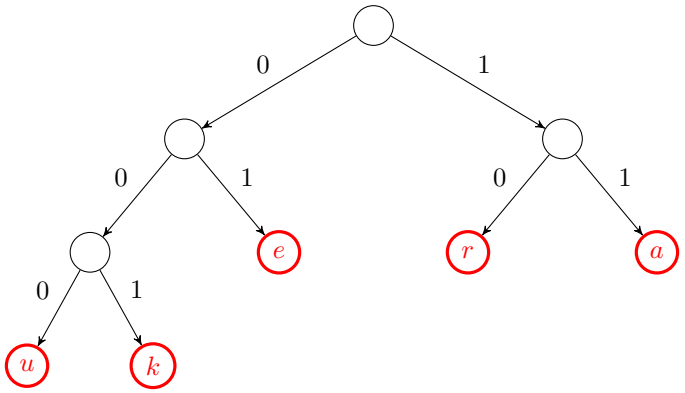
Equivalently, we can represent a prefix code as a tree, where leaves are the symbols and the paths from the root are the corresponding encoding.




Hence, an equivalent definition for average bits per symbol is

$$\text{ABL}(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x).$$

Note that this tree (or prefix code) is not optimal. We can improve it by pushing $u$ one up, or equivalently, use $u \mapsto 000$ instead of 0000. Doing this reduces the depth of $u$ by 1, and does not change the depth of any other symbol, thus decreasing ABL by $1 \cdot f_u = 0.05$. Therefore, the new ABL will be $2.3 - 0.05 = 2.25$ bits/symbol, for which the tree is as follows:




**Definition 13.4.** A tree is **full** if every non-leaf node has exactly 2 children.

**Proposition 13.5.** The tree corresponding to an optimal prefix code is full.

*Proof.* Assume by contradiction that $T$ is the tree of an optimal prefix code and is not full. Therefore, there is a node, denoted $y$, that has only one child. Then we just remove $y$ and connect its child to its parent. This reduces the depth of all the children of $y$ by 1 and does not increase the depth of any other node. This reduces ABL, in contradiction to the assumption that $T$ is optimal. Hence we are done. □

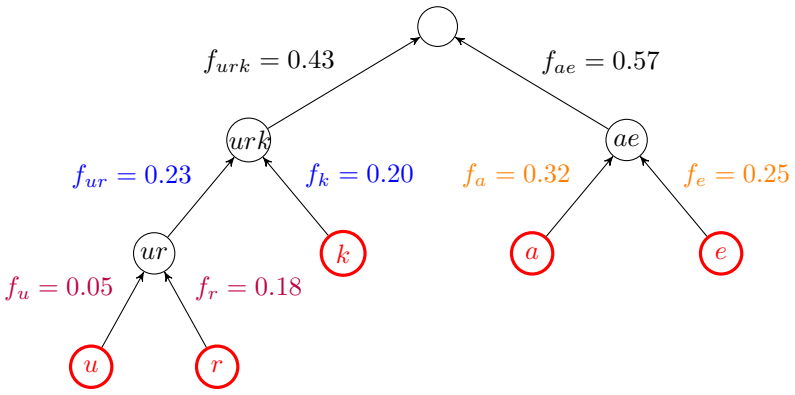
### 13.1.2 Greedy Algorithm

**Greedy Approach.** In an optimal tree, the lowest level must contain the symbols of the lowest frequencies (Observation 1). Otherwise, we can switch with a leaf in a higher level to reduce ABL. Also, the lowest level must contain at least two sibling leaves (Observation 2). Otherwise, the tree is not full and thus not optimal. Finally, the order in which the symbols appear in a level does not matter (Observation 3).

**Proposition 13.6.** There is an optimal tree where the two symbols of lowest frequencies are in sibling leaves.

*Proof.* First, by Observation 2, we know there are two sibling leaves in the bottom layer. Next, by Observation 1, the two symbols of the lowest frequencies must be in the bottom layer. Finally, using Observation 3, we can shuffle symbols in the bottom layer so the two symbols of the lowest frequencies are siblings. □

Hence, the idea is to construct the tree "from leaves." Make two sibling leaves for the two symbols of the lowest frequencies, and recursively build the tree for the remaining symbols plus a "meta symbol" combining the two symbols, and whose frequency is given by their sum. For instance,




| | | | | |
|---|---|---|---|---|
| $f_a = 0.32$ | $f_e = 0.25$ | $f_k = 0.20$ | $f_u = 0.05$ | $f_r = 0.18$ |
| $f_a = 0.32$ | $f_e = 0.25$ | $f_k = 0.20$ | $f_{ur} = 0.23$ | |
| $f_a = 0.32$ | $f_e = 0.25$ | $f_{urk} = 0.43$ | | |
| $f_{ae} = 0.57$ | | $f_{urk} = 0.43$ | | |

**Proposition 13.7.** Following the scheme above, we have in each step

$$\mathrm{ABL}(T') = \mathrm{ABL}(T) - f_w,$$

where $w$ is the "meta symbol" combining the symbols $y$ and $z$, $T$ is the original tree, and $T'$ is the modified tree.

*Proof.* $T'$ is like $T$, except that $y$ and $z$ are one level up, and note that $f_w = f_y + f_z$. □

**Theorem 13.8.** The **Huffman code**, as introduced above, achieves the smallest ABL.

*Proof.* We prove by induction on $n = |S|$. For the base case when $n = 2$, obviously Huffman code is optimal with ABL $= 1$. Then we assume that the statement is true for $(n-1)$ and prove for $n$. Assume by contradiction that there exists a tree $Z$ with $\mathrm{ABL}(Z) < \mathrm{ABL}(T)$. By Proposition 13.6, we can assume that $y$ and $z$ are siblings in $Z$. Let $Z'$ be obtained from $Z$ by merging $y$ and $z$ into their parent (*i.e.*, exactly like how $T'$ is obtained from $T$). But then,

$$\mathrm{ABL}(Z') = \mathrm{ABL}(Z) - f_w, \qquad \mathrm{ABL}(T') = \mathrm{ABL}(T) - f_w,$$

and thus $\mathrm{ABL}(Z') < \mathrm{ABL}(T')$, contradicting the assumption that Huffman code is optimal for $(n-1)$ symbols. The inductive step is done. □

**Algorithm.** A simple implementation of the above greedy approach is shown in Algorithm 19.

**Algorithm 19** HUFFMAN($n$)

```
1: if |S| = 2 then
2:     return a tree with the root and the two leaves;
3: end if
4: y, z ← the symbols of the lowest frequencies in S;
5: w ← the combined symbol of y and z with frequency f_w = f_y + f_z;
6: S' = S\{y, z} ∪ {w};
7: T' = HUFFMAN(S');
8: return the tree with two additional children y and z of leaf w in T';
```

**Runtime.** Naively, Line 4 takes $\Theta(n)$ time, so $T(n) = T(n-1) + \Theta(n)$, which gives $T(n) = \Theta(n^2)$. A better way is to store the symbols in a heap or priority queue, allowing us to implement Lines 4, 5, and 6 in only $\Theta(\log n)$ time. Hence $T(n) = T(n-1) + \Theta(\log n)$, which gives $T(n) = \Theta(n \log n)$.

# 14 11/2 Lecture

## 14.1 Graphs

### 14.1.1 Categorization

There are **undirected graphs** and **directed graphs**. In undirected graphs, each edge is represented by the set of two nodes. The order does not matter, and no self-loop is allowed. In directed graphs, each edge is represented by the tuple of two nodes. The order matters, and self-loops are usually allowed.

**Remark 14.1.** There should be no parallel edges.

### 14.1.2 Applications

- Navigation/maps: directed graphs of places and roads.
- Google Pagerank: directed graph of websites (vertices) and links (edges).
- Social network: (un)directed graph of friends. 3 billion vertices and approximately 300 billion edges. 4.5 is the average distance between two random vertices.
- Internet: routing in graph of computers and connections (BGP).

- Biological networks (protein-protein interaction, approximately 10000 vertices).

Example: Consider a pocket cube ($2 \times 2 \times 2$ Rubik's cube). In the configuration graph, each vertex stands for a possible configuration. The number of vertices should be

$$\#\text{ vertices} = |V| = \underbrace{8!}_{\text{permutation of cubies}} \times \underbrace{3^8}_{\text{twists of cubies}} \div \underbrace{24}_{\text{all rotation of the cube}} \div \underbrace{3}_{\text{reachable part}} = 264539520.$$

### 14.1.3 Representation

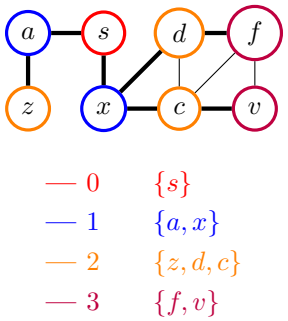
**Adjacency List.** This is the most common graph representation. An adjacency list is an array Adj of $|V|$ linked lists. For each $u \in V$, Adj$[u]$ stores all of $u$'s neighbors, *i.e.*, Adj$[u] = \{v \in V \mid (u, v) \in E\}$. The space complexity to store a graph by adjacency list is $\Theta(|V| + |E|)$.

**Adjacency Matrix.** This is not very common for graph representation. An adjacency matrix $A$ is a 2-dimensional array where $A[u][v] = 1$ if and only if $(u, v) \in E$, and otherwise 0. The space complexity to store a graph by adjacency matrix is $\Theta(|V|^2)$.

## 14.2 Breadth First Search (BFS)

### 14.2.1 Overview

Input: a graph $G = (V, E)$, either directed or undirected, given as an adjacency list and a "source" vertex $s$.
Output: all vertices reachable from $s$.
Runtime: $O(|V| + |E|)$ (linear time).




The BFS tree allows us to find the shortest path to $s$ from any vertex $u$, as is shown by the thick edges. These are the edges that discover some node for the first time in the BFS.

### 14.2.2 Algorithm

The implementation of BFS is as shown in Algorithm 20.

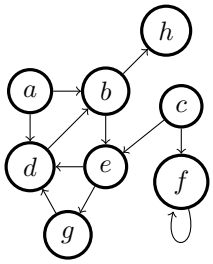
**Algorithm 20** BFS(Adj, $s$)

```
frontier ← [s];
s.color ← ■;
for u ∈ V\{s} do
    u.color ← □;                                   ▷ Initialize other vertices
end for
while frontier ≠ ∅ do
    next ← [];                                     ▷ Next frontier
    for u ∈ frontier do
        for v ∈ Adj[u] do                          ▷ All neighbors of u
            if v.color = □ then
                v.color = ■;
                next.append(v);
            end if
        end for
    end for
    frontier ← next;
end while
```

# 15 11/9 Lecture

## 15.1 Depth First Search (DFS)

### 15.1.1 Overview




- Unlike BFS, DFS typically does **not** provide shortest paths.
- We usually use DFS to learn something about the whole graph. As a result, we usually make sure the entire graph is explored.
- DFS is usually used for directed graphs, but it also works for undirected graphs.

### 15.1.2 Algorithm

The implementation of DFS is as shown in Algorithm 21.

**Algorithm 21** DFS(Adj)

```
for u ∈ V do
    u.color ← □;                                   ▷ Initialization
end for
for u ∈ V do
    if  then
        DFSVISIT(Adj, u);                          ▷ Algorithm 22
    end if
end for
```

**Algorithm 22** DFSVISIT(Adj, u)

```
1: u.color ← ■;
2: for v ∈ Adj[u] do
3:     if v.color = □ then
4:         DFSVISIT(Adj, v);
5:     end if
6: end for
7: u.color ← ■;
```

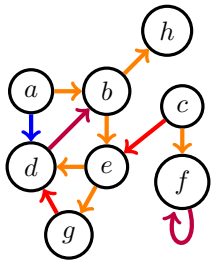
### 15.1.3 Runtime

We only call DFSVISIT (Algorithm 22) once per vertex (when it is white and first discovered). Therefore, the total time in DFSVISIT is

$$\sum_{v \in V} (\Theta(1) + \Theta(|\text{Adj}[v]|)) = \Theta(|V|) + \Theta\left(\sum_{v \in V} |\text{Adj}[v]|\right) = \Theta(|V| + |E|).$$

This is thus linear runtime (with respect to the size of input), just like BFS.

## 15.2 Edge Classification




- **Tree edge →:** the edges used to visit a new vertex. In DFS, $(u, v)$ is a tree edge if $v$ is □ when visited.
- **Forward edge →:** the edges from vertex to descendant in the tree (but not child). In DFS, $(u, v)$ is a tree edge if $v$ is ■ when visited, and $v$ is discovered after $u$.
- **Backward edge →:** the edges from vertex to ancestor. In DFS, $(u, v)$ is a tree edge if $v$ is ■ when visited.
- **Cross edge →:** the edges between non-ancestor vertices (basically everything else). In DFS, $(u, v)$ is a tree edge if $v$ is ■ when visited, and $v$ is discovered (and in fact finished) before $u$.

# 16 11/14 Lecture

## 16.1 White-Path Theorem

**Theorem 16.1.** In DFS of undirected graphs, every edge is either a tree edge or a backward edge (*i.e.*, there are no forward or cross edges). In other words, when we first explore an edge $(u, v) \in E$, $v$ cannot be ■.

*Proof.* If $v$ is ■, it means that we finished exploring its neighbors. In particular, we should have explored that edge from $v$ (and it would become a backward edge or a tree edge). □

**Proposition 16.2.** An undirected graph is **acyclic** (*i.e.*, no cycles) if and only if there are no backward edges.

*Proof.* If there are no backward edges, then all edges are tree edges by the previous theorem, so the graph is acyclic. To prove the converse, if there is a backward edge, it closes a cycle. □

**Theorem 16.3** (White-path theorem)**.** $v$ is a descendant of $u$ in the DFS forest if and only if at time $u$ is discovered, there exists a path from $u$ to $v$ with only □ vertices.

*Proof.* Assume $v$ is a descendant of $u$ in a DFS forest. Then there is a path from $u$ to $v$. Clearly all the vertices on this path are discovered in order, so $u$ must be the first to be discovered among them. To prove the converse, $v$ is detected during the exploration from $u$ and thus a descendant of it. □

# 17 11/16 Lecture

## 17.1 Directed Acyclic Graphs (DAG)

**Lemma 17.1.** A directed graph is acylic if and only if DFS finds no back edges.

*Proof.* Equivalently, we will prove that a directed graph has cycles if and only if DFS finds a back edge. Indeed, any back edge immediately creates a cycle. To prove the converse, assume there is a cycle. Let $u_0$ be the first vertex of the cycle discovered by DFS, and label the remaining vertices as $u_1, u_2, \cdots, u_k$ in this order. Then, by the white-path theorem, $u_k$ will be a descendant of $u_0$. Then the edge $(u_k, u_0)$ is a back edge. □

**Remark 17.2.** By the previous lemma, we know that we can identify if a graph has cycles in time $\Theta(|V| + |E|)$ using DFS.

## 17.2 Topological Sort

### 17.2.1 Overview

**Topological sort** takes a directed graph and returns an array of the vertices where each vertex appears before all the vertices it points to.
One important thing to mention: whenever you see a directed acyclic graph, think of topological sort first!

### 17.2.2 Algorithm

We introduce the following topological sorting algorithm TopologicalSort (Algorithm 23) based on DFS (Algorithm 21).

**Algorithm 23** TopologicalSort($G$)

```
result ← [];
Apply DFS traversal, whenever v becomes ■, result.insert(0, v);
return result;
```

### 17.2.3 Correctness

**Proposition 17.3.** TopologicalSort outputs a topologically-sorted array of vertices.

*Proof.* The goal is to show that for any edge $(u, v)$, $v$ is finished before $u$ in a directed graph. When we explore $(u, v)$, $v$ is either ■ or □. If $v$ is ■, since $u$ is still active, $u$ must be finished after $v$. If $v$ is □, we will recursively visit $v$. When the recursive call returns, $v$ will be finished, but at this time $u$ is still active, so again $u$ must be finished after $v$. □

**Remark 17.4.** Note that topological sort is well-defined only for directed acyclic graphs, since otherwise there will be at least one vertex breaking the topological order.

### 17.2.4 Runtime

The runtime is $\Theta(|V| + |E|)$, same as DFS. Note that we do not really need to sort by the finishing time of the vertices. This is automatically done during the DFS traversal.

## 17.3 Strongly-Connected Components (SCC)

### 17.3.1 Overview

A directed graph is called **strongly connected** if there is a path in each direction between each pair of vertices of the graph. For a directed graph $G$, we combine each of its strongly connected component as a supernode, and we connect two supernodes if some vertex of one supernode connects to some vertex of the other. The resulting graph is denoted as $G^{\text{SCC}}$.

**Proposition 17.5.** For all directed graphs $G$, $G^{\text{SCC}}$ is acyclic.

*Proof.* A cycle in $G^{\text{SCC}}$ would form a larger strongly connected component, resulting in a contradiction. □

Consider a DFS traversal on $G$. We can pretend that this DFS takes place on $G^{\text{SCC}}$. We say that

- A component is discovered and becomes ■ the first time one of its vertices is discovered.
- A component is finished and becomes ■ when all its vertices are finished and become ■.

**Lemma 17.6.** The "virtual walk" on $G^{\text{SCC}}$ induced by a DFS on $G$ is a valid DFS traversal on $G^{\text{SCC}}$.

*Proof.* When a component $C$ is first discovered, say by a call to DFSVISIT($u$) for some $u \in C$, then when that call returns, all vertices in $C$ are finished by the white path theorem, just like in an actual DFS traversal. □

Therefore, if there is an edge from $C$ to $C'$ in $G^{\text{SCC}}$ (which is a DAG), then the finish time of $C$ is greater than the finish time of $C'$, just like in the analysis of a topological sort. Hence, the component with the largest finish time must be first in the topological sorting of $G^{\text{SCC}}$. As a result, the last vertex to finish is in the first component of $G^{\text{SCC}}$. We can identify the entire first component by running DFSVISIT on the transpose graph $G^T$ (with all edges in $G$ flipped), starting from that vertex.

#### 17.3.2 Algorithm

We introduce the following algorithm SCC (Algorithm 24) for finding the strongly connected components (SCC) in a directed graph.

**Algorithm 24** SCC($G$)

1: Call DFS($G$) to compute the finish time of all vertices; ▷ Algorithm 21
2: Call DFS($G^T$), where in the main outer loop, we consider vertices in an decreasing order with respect to the finish time; ▷ Algorithm 21
3: Output each tree in the DFS forest found in the Line 2 as an SCC component;

#### 17.3.3 Runtime

The runtime of SCC (Algorithm 24) is $\Theta(|V|+|E|)$. Each DFS traversal runs $\Theta(|V|+|E|)$, and the construction of the $G^T$ takes $\Theta(|E|)$. Therefore, the total runtime is just $\Theta(|V|+|E|)$.

# 18 11/21 Lecture

## 18.1 Minimum Spanning Trees (MST)

### 18.1.1 Overview

Suppose $G$ is an undirected, connected graph with weight $w(u,v)$ associated to each edge. A **spanning tree** is a subset $T \subseteq E$ that is a tree (*i.e.*, acyclic and connected) that have the same set of vertices as $G$. The goal is to find a spanning tree $T$ minimizing the value

$$w(T) = \sum_{(u,v)\in T} w(u,v).$$

A generic approach is to start with an empty tree $A$ and maintains the loop invariant that $A$ is always a subset of some MST. Therefore, until $A$ spans $G$, we iteratively finds a **safe** edge $(u,v) \in E$ and add it to $A$. By the loop invariant, there must exist some safe edge in each iteration.

**Definition 18.1.** A **cut** of $G$ is a partition of $V$ into $(S, V\backslash S)$. An edge $(u,v)$ **crosses** a cut $(S, V\backslash S)$ if $u \in S$ and $v \in V\backslash S$ or vice versa. A cut **respects** $A$ if no edge of $A$ crosses the cut. Finally, $(u,v)$ is a **light edge** crossing a cut if it has minimum weight among all edges crossing that cut.

**Theorem 18.2.** Let $A \subseteq E$ be a subset of some MST, and $(S, V\backslash S)$ be a cut respecting $A$. Then any light edge $(u,v)$ crossing that cut is **safe** for $A$.

*Proof.* Let $T$ be an MST containing $A$. If $(u, v) \in T$, then we are done. Otherwise, $(u, v) \notin T$, then since $T$ is an MST, there must be a path from $u$ to $v$ in $T$. Because $u \in S$ and $v \in V \backslash S$, there exists an edge $(x, y)$ on that path that crossed the cut. Given that $(u, v)$ is a light edge, we have that

$$w(u, v) \leq w(x, y).$$

Consider $T' = T \backslash \{(x, y)\} \cup \{(u, v)\}$. First, $T'$ is a spanning tree. Second, we have that

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T),$$

and in fact, equal, since $T$ is an MST as well. Therefore, $T'$ is also an MST and so $(u, v)$ is safe. □

### 18.1.2 Kruskal's Algorithm

Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge $(u, v)$ of the least weight (except those that will form a cycle).

**Algorithm.** The algorithm MSTKruskal (Algorithm 25) as follows is an implementation of the Kruskal's algorithm using a disjoint-set data structure to maintain several disjoint sets of elements (Section A.1).

**Algorithm 25** MSTKruskal$(G, w)$

```
1: A ← ∅;
2: for v ∈ V do
3:     MakeSet(v);
4: end for
5: Sort E in a non-decreasing order with respect to weight;
6: for (u, v) ∈ E do                          ▷ Starting from the lighter edges
7:     if FindSet(u) ≠ FindSet(v) then        ▷ u and v are in different connected components
8:         A ← A ∪ {(u, v)};
9:         Merge(u, v);
10:    end if
11: end for
```

**Correctness.** Let $C_1$ and $C_2$ denote the two trees that are connected by $(u, v)$. Since $(u, v)$ must be the light edge connecting $C_1$ to some other tree, clearly $(u, v)$ is a safe edge for $C_1$.

**Runtime.** Making the disjoint sets takes $|V| \cdot \Theta(1) = \Theta(|V|)$ time. Sorting $E$ in non-decreasing order takes $\Theta(|E| \log |E|)$ time. The FindSet operations takes $|E| \cdot \Theta(\log |V|) = \Theta(|E| \log |V|)$ time. The Merge operation happenss for exactly $|V| - 1$ times since a spanning tree has $|V| - 1$ edges, so this takes $(|V| - 1) \cdot \Theta(\log |V|) = \Theta(|V| \log |V|)$ time. Therefore, the total runtime of MSTKruskal (Algorithm 25) is $\Theta(|V| + |E| \log |E| + |E| \log |V| + |V| \log |V|)$ time. Clearly, we do not need to consider the first summand since it is subject to the last summand. We do not need to consider the last summand either, since given $G$ is connected, we have that $|E| \geq |V| - 1$, so it must be subject to the third summand. In an undirected graph, $|E| \leq \binom{|V|}{2}$, so $|E| = O(|V|)^2$, implying that the second term and the third term has the same order. Therefore, we conclude that the runtime of MSTKruskal (Algorithm 25) can be simplified as $\Theta(|E| \log |V|)$.

**Remark 18.3.** There exists a super fast disjoint-set data structure that can perform the loop over $E$ in $O(|E| \cdot \alpha |V|)$ time, where $\alpha$ represents the *inverse Ackermann funtion* which grows extremely slowly. However, the sorting still runs $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$, so the asymptotic bound for the total runtime does not change unless edges are already sorted.

### 18.1.3 Prim's Algorithm

Prim's algorithm grows a spanning tree starting from some vertex $r$, and at each step, adds the vertex outside the tree that is connected by the lightest edge to the tree.

**Algorithm.** The algorithm MSTPrim (Algorithm 26) as follows is an implementation of the Prim's algorithm using a minimum priority queue structure.

**Algorithm 26** MSTPrim$(G, w, r)$

**Require:** $r$ is the vertex to start with.

```
for v ∈ V do
    v.key ← ∞;
    v.val ← null;
end for
r.key ← 0;
Q ← PriorityQueue(V);                                   ▷ Q is the waiting list
while Q ≠ ∅ do
    v ← ExtractMin(Q);
    for u ∈ Adj[v] do
        if u ∈ Q and w(u, v) < u.key then
            u.val = v;
            DecreaseKey(Q, u, w(u, v));
        end if
    end for
end while
return {(u.val, u); u ∈ V\{r}};
```

**Correctness.** Each edge we add is a light edge for the cut defined by my current tree, and thus a safe edge.

**Runtime.** The initialization, including PriorityQueue (which is based on the construction of a minimum binary heap), takes $\Theta(|V|)$ time. ExtractMin is performed $\Theta(|V|)$ times, which each operation taking $O(\log|V|)$ time. DecreaseKey is performed $\Theta(|E|)$ times, which each operation taking $O(\log|V|)$ time. Therefore, the total runtime of the algorithm MSTPrim is $O(|V| + |V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$.

**Remark 18.4.** Using priority queues based on Fibonacci heaps, we get ExtractMin in $O(\log|V|)$ time as before, but DecreaseKey in amortized $O(1)$ time. Therefore, the total runtime will become $O(|V|\log|V| + |E|)$, typically better than Kruskal's algorithm.

# 19 11/30 Lecture

## 19.1 Single Source Shortest Paths (SSSP)

### 19.1.1 Overview

Input: a directed graph $G = (V, E)$ with non-negative weights $w : E \to [0, \infty)$, and a source vertex $s \in V$.
Goal: find shortest paths from $s$ to all other vertices.

**Remark 19.1.** We can also consider negative edges, and there are algorithms like Bellman-Ford running in time $O(|V| \cdot |E|)$.

**Remark 19.2.** Even if we just care about the shortest path between $s$ and some other vertex $t$, the running time is asymptotically the same.

### 19.1.2 Dijkstra's Algorithm

**Algorithm.** The algorithm Dijkstra (Algorithm 27) as follows is an implementation of the Dijkstra's algorithm using a minimum priority queue structure.

**Algorithm 27** DIJKSTRA$(G, w, s)$

```
1: for v ∈ V do
2:     v.key ← ∞;
3:     v.val ← null;
4: end for
5: s.key ← 0;
6: Q ← PriorityQueue(V);
7: while Q ≠ ∅ do
8:     v ← ExtractMin(Q);
9:     for u ∈ Adj[v] do
10:        if u.key > v.key + w(u, v) then
11:            u.val = v;
12:            DecreaseKey(Q, u, v.key + w(u, v));
13:        end if
14:    end for
15: end while
```

**Correctness.** Dijkstra's algorithm is correct for directed graphs with non-negative edges.

**Theorem 19.3.** Dijkstra's algorithm DIJKSTRA (Algorithm 27) is correct.

*Proof.* Let $\delta_s(v)$ be the weight of the shortest path from $s$ to $v$. Then we prove by induction on $n = |V| - |Q|$ that $v.\text{key} = \delta_s(v)$ for all $v \in V \backslash Q$.

**Base case ($n = 1$).** Since $Q$ only shrinks in size, the only time $n = 1$ is when $V \backslash Q = \{s\}$, so $s.\text{key} = \delta_s(s) = 0$, which is indeed correct.

**Inductive step.** Let $u$ be the last vertex extracted from $Q$, and let $Q' = Q \backslash \{u\}$. Assume that for each $v \in V \backslash Q$, we have that $v.\text{key} = \delta_s(v)$. Then we prove for the statement for $Q'$. By the inductive hypothesis, it suffices to prove that $u.\text{key} = \delta_s(u)$. Assume for contradiction that the shortest path from $s$ to $u$ is $\mathscr{P}$ with $u.\text{key} > W(\mathscr{P}) = \delta_s(u)$. $\mathscr{P}$ starts in $V \backslash Q'$ and at some time enters $Q'$ (to get to $u \in Q'$). Let $(x, y)$ be the first edge along $\mathscr{P}$ that enters $Q'$. Let $\mathscr{P}_x$ be the subpath of $\mathscr{P}$ from $s$ to $x$. Clearly, we have that

$$W(\mathscr{P}_x) + w(x, y) \leq W(\mathscr{P}).$$

Moreover, by the inductive hypothesis, $x.\text{key} = \delta_s(x)$, so that $x.\text{key} \leq W(\mathscr{P}_x)$, giving that

$$y.\text{key} \leq x.\text{key} + w(x, y) \leq W(\mathscr{P}_x) + w(x, y) \leq W(\mathscr{P}).$$

But since $u$ was picked by DIJKSTRA, we must have $u.\text{key} \leq y.\text{key} \leq W(\mathscr{P})$, leading to a contradiction. Therefore, we must have that $u.\text{key} \leq \delta_s(u)$. But still, $u.\text{key} \geq \delta_s(u)$ which is a loop invariant, so the inductive step is done. By mathematical induction, we can conclude that $v.\text{key} = \delta_s(v)$ for all $v \in V \backslash Q$ for all possible $Q$ during the execution of DIJKSTRA. Taking $Q = \varnothing$ when DIJKSTRA terminates, we conclude our proof. □

**Runtime.** The runtime analysis of DIJKSTRA (Algorithm 27) is identical to that of MSTPRIM (Algorithm 26), except that we can no longer assume $|E| \leq |V| - 1$. Therefore, the runtime with a binary heap implementation would be $O((|V| + |E|) \log |V|)$, and the runtime with a Fibonacci heap would be amortized $O(|V| \log |V| + |E|)$.

# A Appendix

## A.1 Disjoint-Set Data Structures

A **disjoint-set data structure** represents a dynamic collection of pairwise disjoint sets $\mathscr{S} = \{S_1, \cdots, S_r\}$. Given an element $u$, we denote by $S_u$ the set containing $u$. We will equip each set $S_i$ with a representative element $\text{rep}[S_i]$. The disjoint-set data structure supports the following operations:

- MakeSet($u$): creates a new set containing the single element $u$.
- FindSet($u$): returns the representative $\text{rep}[S_u]$.
- Merge($u, v$): replaces $S_u$ and $S_v$ with $S_u \cup S_v$ in $\mathscr{S}$, and updates representative elements as appropriate.

This data structure is best implemented using a forest-of-trees implementation.

- MakeSet($u$): initializes a new tree with root node $u$. This takes $\Theta(1)$ time.
- FindSet($u$): walks up the corresponding tree from $u$ to the root (which is the representative element of that tree). This takes $\Theta(\log n)$ time if the tree is balanced.
- Merge($u, v$): changes $\text{rep}[S_v]$'s parent to $\text{rep}[S_u]$ (in fact, we may do the converse, based on which makes the resulting tree shorter). This takes $\Theta(1) + 2t_{\text{FindSet}}$ time, which is $\Theta(\log n)$ times if the tree is balanced.