INFORMATION & COMPUTER SCIENCES
UNIVERSITY of HAWAIʻI at MĀNOA

Ch 3.1: Algorithms

ICS 141: Discrete Mathematics for Computer Science I

KYLE BERNEY
DEPARTMENT OF ICS, UNIVERSITY OF HAWAII AT MANOA

Algorithms

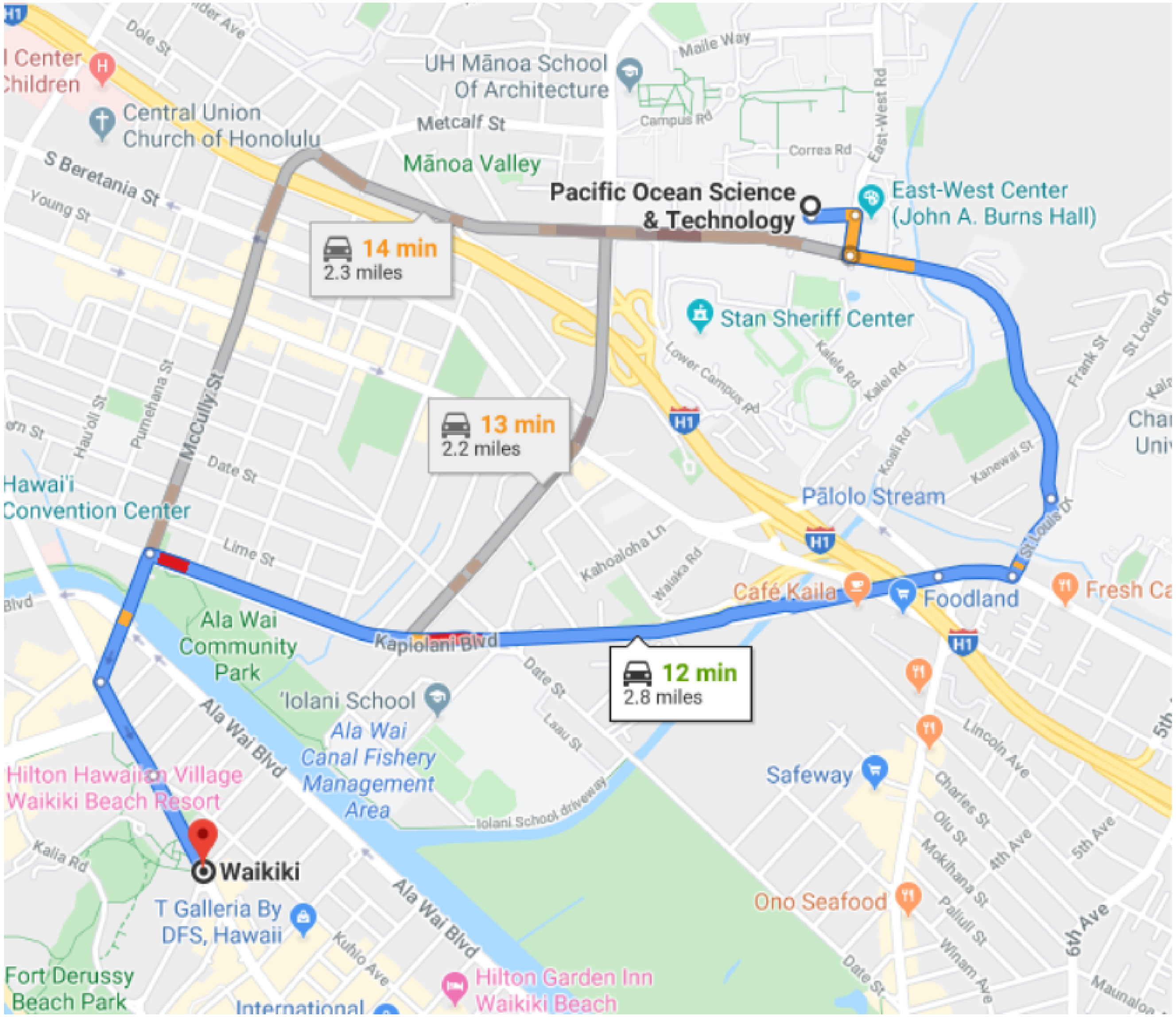
- Definition: An algorithm is a finite sequence of unambiguous (simple) instructions for performing a computation or solving a problem
- Ex:
  - Directions
  - Cooking recipes
  - Everyday actions
    - Tying your shoes
    - Folding clothes
  - Organization
    - Sorting playing cards
  - Routines
    - Exercise routines
    - Shower routines

Algorithms

- Describing algorithms
  - Visual representations

Algorithms

- Ex. Directions from POST building to Waikiki



Algorithms

- Describing algorithms
  - Visual representations
  - Everyday language

Algorithms

- Ex: Finding the largest element in a finite sequence of integers.
  1. Set the current maximum value equal to the first integer in the sequence
  2. Compare the next integer in the sequence and the current maximum value
     - If the next integer is larger than the current maximum, then set the current maximum equal to this integer
  3. If there are more integers in the sequence, repeat step 2.
  4. After there are no integers left in the sequence, the current maximum value will be set to the largest integer in the sequence

Algorithms

- Describing algorithms
  - Visual representations
  - Everyday language
  - A computer language (i.e., a programming language)

Pseudocode

- Instead of chosing a particular programming language, we use pseudocode
- Pseudocode:
  - Resembles programming languages, but is intended to be human readable
  - Focuses on logic, rather than syntax
  - Bridges the gap between problem-solving and coding
- There is not a strict standard for how to write pseudocode
  - Should be clear and unambiguous

Algorithms

- Ex: Finding the largest element in a finite sequence of integers.

$\textsc{Max}(A[1 \ldots n])$

$\quad max = A[1]$

$\quad \textbf{for} \;\; i = 2 \;\; \textbf{to} \;\; n$

$\qquad \textbf{if} \;\; max < A[i]$

$\qquad\quad max = A[i]$

$\quad \textbf{return} \; max$

Algorithms

- Ex: Finding the largest element in a finite sequence of integers.

```
MAX(A[1 . . . n])
    max = A[1]
    for i = 2 to n
        if max < A[i]
            max = A[i]
    return max
```

- *Remark:* In math and theoretical computer science, we typically use 1-indexed arrays (rather than 0-indexed arrays)

Algorithms

- Describing algorithms
  - Visual representations
  - Everyday language
  - A computer language (i.e., a programming language)
- *Remark:* Combinations of the above can be used together
  - In textbooks or research papers, algorithms are described with pseudocode, figures, and/or descriptions of the steps
  - When writing code, it is good practice to also include comments describing your code

Properties of Algorithms

- Input: an algorithm has input values from a specified set
- Output: an algorithm produces values from a specified set
- Definiteness: steps of an algorithm are defined precisely
- Correctness: an algorithm should produce the correct output values
- Finiteness: an algorithm should produce the desired output after a finite number of steps for all inputs
- Effectiveness: it is possible to perform each step of an algorithm exactly
- Generality: the algorithm is applicable to all problems of the desired form

Searching Algorithms

- Searching for a particular element in a collection of elements
- Problem: Given an element x and a collection of n elements, $a_1, a_2, \ldots, a_n$, find the location of x or determine that x is not in the collection

Linear Search

- Iterate through the n elements and check whether it is equal to x or not
  - If the current element is equal to x, we return its location and terminate the algorithm
- If all elements were inspected and x has not been found, then we return that x was not found and terminate the algorithm

```
LINEARSEARCH(A[1 . . . n], x)
    for i = 1 to n
        if x == A[i]
            return i
    return NOT FOUND
```

Binary Search

- <u>Precondition:</u> The collection of elements is sorted (typically in increasing order)
- Compare x with the median (i.e., middle) element
  - If the median element is x, we return its location
  - If the median element is greater than x, we continue the algorithm only on elements that are smaller than the median element
  - If the median element is smaller than x, we continue the algorithm only on elements that are greater than the median element

Binary Search

$\textsc{BinarySearch}(A[1 \ldots n], x)$

```
left = 1
right = n
while left ≤ right
    mid = ⌊(left + right)/2⌋
    if x == A[mid]
        return mid
    else if x < A[mid]
        right = mid − 1
    else
        left = mid + 1
return NOT FOUND
```

Binary Search

- *Question:* Why is $mid = \lfloor (left + right)/2 \rfloor$?
  - The number of elements, denoted n, contained in $A[left \dots right]$ is:
$$n = right - left + 1$$
  - (Lower) median, denoted k, is defined as:
$$k = \left\lfloor \frac{n+1}{2} \right\rfloor$$
  - k-th element starting from the index $left$

$$\begin{aligned}(left - 1) + \left\lfloor \frac{n+1}{2} \right\rfloor &= \left\lfloor \frac{2left - 2 + (right - left + 1) + 1}{2} \right\rfloor \\ &= \left\lfloor \frac{right + left}{2} \right\rfloor .\end{aligned}$$

Sorting Algorithms

- Ordering elements in a collection of elements
- <u>Problem:</u> Given a way to order elements (i.e., a way to compare two elements) and a collection of n elements $a_1, a_2, \ldots, a_n$, rearrange the collection into $a'_1, a'_2, \ldots, a'_n$ such that

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Bubble Sort

1. Iterate through the array and compare adjacent elements
   - If the adjacent elements are out of order, swap their positions
2. Repeat step 1. $(n-2)$ additonal times (i.e., in total $(n-1)$ executions of step 1. are performed)

- *Intuition:* after every execution of step 1., the larger elements are ”bubbled” to the end of the array
  - After the first pass, the largest element is in $A[n]$
  - After the second pass, the second largest element is in $A[n-1]$
  - After the third pass, the third largest element is in $A[n-2]$

$\vdots$

Bubble Sort

BubbleSort($A[1 \ldots n]$)
  for $i = 1$ **to** $n - 1$
    for $j = 1$ **to** $n - i$
      if $A[j] > A[j + 1]$
        Swap($j, j + 1$)

Insertion Sort

- *Intuition:* Works similar to how many people sort a hand of playing cards

1. Start with an empty hand of cards
2. Pickup a card one at a time and insert it into the correct position in your hand
   - To find the correct position, compare the card with each card already in your hand
3. Algorithm terminates when all cards have been inserted into your hand

Insertion Sort

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

String Matching

- Asks whether a particular string of characters called the <u>pattern</u>, denoted P, occurs within another string T.
- When the pattern P begins at position $(s + 1)$ in the string T
  - We say that P occurs with <u>shift</u> s in T
- <u>Problem:</u> Given a pattern P and string T, find all valid shifts of P

Naive String Matching

- Given a pattern $P[1 \ldots m]$ and a string $T[1 \ldots n]$
- For each of the $n - m + 1$ possible values of s
  - Check whether

$$P[1 \ldots m] = T[s + 1 \ldots s + m]$$

$\textsc{NaiveStringMatching}(P[1 \ldots m], T[1 \ldots n])$

```
for s = 0 to n − m
    j = 1
    while j ≤ m and T[s + j] == P[j]
        j = j + 1
    if j > m
        PRINT("s is a valid shift")
```

Optimization Problems

- Some problems are concerned with finding a solution that either minimizes or maximizes the value of some parameter
  - Known as optimization problems
  - Parameter that is optimized is called the objective function
- Two common algorithmic approaches:

1. Greedy algorithms
2. Dynamic programming

Greedy Algorithms

- A greedy algorithm always makes the choice that looks "best" at the moment
- Optimization problems can be solved using greedy algorithms if they exhibit:
  - Greedy Choice Property
    - If the objective function is optimized locally, then it is optimized globally
    - The greedy choice is always part of some optimal solution
  - Optimal Substructure
    - An optimal solution to the problem contains optimal solutions to the subproblems

Activity Selection

- Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities.
- Each activity a_i has a start time s_i and a finish time f_i where

$$0 \leq s_i < f_i < \infty$$

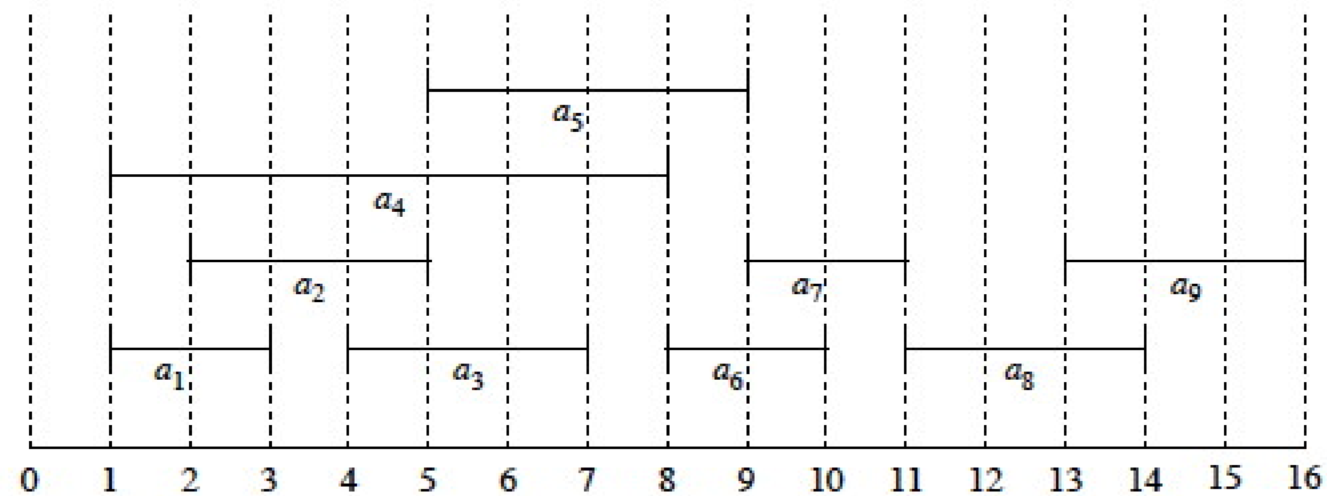
- If selected, activity a_i takes place during the interval $[s_i, f_i)$
- Activities a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
  - In other words, $s_i \geq f_j$ or $s_j \geq f_i$
- Problem: Select a maximum-size subset of S of mutually compatible activies
  - Assume that S is given such that the activities are sorted in increasing order of finish time

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

Activity Selection

- Ex: Consider the following activities:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



- $\{a_1, a_3, a_6, a_8\}$ is an optimal solution
- $\{a_2, a_5, a_7, a_9\}$ is another optimal solution

Activity Selection

- Optimal Substructure
  - An optimal solution to the problem contains optimal solutions to the subproblems

Activity Selection

- Proof: (Sketch)
  - Let $S_{i,j}$ denote the set of activities that start after a_i and end before a_j
  - Let $A_{i,j}$ be an optimal solution for $S_{i,j}$ which includes some activity a_k
  - Now have two subproblems:
    1. Find mutually compatible activities in $S_{i,k}$
    2. Find mutually compatible activities in $S_{k,j}$
  - Define optimal solutions to the subproblems:
    1. Let $A_{i,k} = A_{i,j} \cap S_{i,k}$
    2. Let $A_{i,k} = A_{i,j} \cap S_{i,k}$

Activity Selection

- Proof: (Sketch)
  - Optimal solution $A_{i,j}$ can be defined as:
    $$A_{i,j} = A_{i,k} \cup \{a_k\} \cup A_{k,j}$$
  - And the number of activities in the optimal solution is
    $$|A_{i,j}| = |A_{i,k}| + 1 + |A_{k,j}|$$
  - "Cut-and-paste" argument
    - Without loss of generality, assume some suboptimal solution to the subproblem $S_{i,k}$, denoted $A'_{i,k}$ is used instead of the optimal solution $A_{i,k}$
    - Since $|A'_{i,k}| < |A_{i,k}|$, it contradicts the assumption that $A_{i,j}$ is the optimal solution since we can always subtitute $A_{i,k}$ for $A'_{i,k}$ and obtain a better solution.

Activity Selection

- Greedy Choice Property
  - If the objective function is optimized locally, then it is optimized globally
  - The greedy choice is always part of some optimal solution
- Greedy Choice:
  - The more time left after running an activity, the more subsequent activities we can fit into the schedule
  - If we choose the first activity to finish, then the most time will be left
  - Since activities are sorted by fnish time, we always start with a_1 then solve the optimization problem for the remaining time

Activity Selection

- <u>Theorem:</u> Let S_k be the set of all activities that start after a_k finishes. If S_k is non-empty and a_m has the earliest finish time in S_k, then a_m is included in some optimal solution

Activity Selection

- Proof: (Sketch)
  - Let A_k be an optimal solution to S_k and let $a_j \in A_k$ have the earliest finish time in A_k
  - If $a_j = a_m$, then we are done
  - Otherwise, let $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ (subtitute a_m for a_j)
  - Since a_j is the first activity to finish in A_k and a_m is the first activity to finish in S_k
$$f_m \leq f_j$$
  - Hence, all activities in A'_k are disjoint and is a valid solution to S_k
  - Moreover, $|A_k| = |A'_k|$, therefore A'_k is also an optimal solution and it includes a_m

Activity Selection

GreedyActivitySelector($S[1 \ldots n], F[1 \ldots n]$)

$A = \{a_1\}$
$k = 1$
for $m = 2$ **to** n
  if $S[m] \geq F[k]$
    $A = A \cup \{a_m\}$
    $k = m$
return A