CSCI-UA 480.4: APS
Algorithmic Problem Solving

Sorting and Searching

Instructor: Joanna Klukowska

created based on materials for this class by
Bowen Yu and materials shared by the authors of
the textbook Steven and Felix Halim

Sorting

Sorting Algorithms

- swap based sorts
    - swapping consecutive elements, $O(N^2)$,
        - bubble sort
        - insertion sort
        - selection sort
    - swapping non-consecutive elements, $O(N \log N)$
        - merge sort
        - quick sort
        - heap sort
    - *Note*: It is not possible to sort N elements using comparisons in time better than $O(N \log N)$

Sorting Algorithms

- counting sort
    - works in $O(N)$
    - assumes that elements to be sorted are in a fixed range of 0 to c

Example

Data to sort: A = [1, 3, 6, 9, 9, 3, 5, 9]

Sorting Algorithms

- counting sort
    - works in $O(N)$
    - assumes that elements to be sorted are in a fixed range of 0 to c

Example

Data to sort: A = [1, 3, 6, 9, 9, 3, 5, 9]

- create an array C with indexes 0 to 9, initialize all values to zero
- for each element i in A
    - increment C[A[i]] by one

```
index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
value  | 0 | 1 | 0 | 2 | 0 | 1 | 1 | 0 | 0 | 3 |
```

- create and empty array B
- for each element i in C
    - if C[i] > 0, add i to B C[i] times

B is the sorted version of A

(of course, this could be done *in place* without using a B array, but we do need to use a C array)

Sorting in C++ and Java

- both languages provide sorts implemented in the libraries
- in both cases the implementations provide $O(N \log N)$ performance
- **but** learn how to implement a comparison operations that can be given to those sorts to decide/alter the sorting order of objects

Challenge: Unique or Not

Task:
given an array A of N integers (N can be very large), determine if all values are unique (i.e., no value appears twice).

Challenge: Unique or Not

Task:
given an array A of N integers (N can be very large), determine if all values are unique (i.e., no value appears twice).

Brute force solution, $O(N^2)$

- for i in 0 .. N-1
    - for j in i+1 .. N-1
        - if A[i] is equal to A[j] return NOT_UNIQUE
- elements are UNIQUE

Challenge: Unique or Not

Task:
given an array A of N integers (N can be very large), determine if all values are unique (i.e., no value appears twice).

Brute force solution, $O(N^2)$

- for i in 0 .. N-1
  - for j in i+1 .. N-1
    - if A[i] is equal to A[j] return NOT_UNIQUE
- elements are UNIQUE

Better, using sorting, $O(N \log N)$

- sort the array (using $O(N \log N)$ sort, of course)
- for i in 1 .. N-1
  - if A[i] is equal to A[i-1] return NOT_UNIQUE
- elements are UNIQUE

Challenge: Restaurant Problem

Task:
At the end of the day a restaurant owner tries to determine what time was the most popular during the evening. The restaurant keeps track of exact time of arrival and departure of each party (assume all parties always arrive and leave together). What was the largest number of parties at the restaurant during that evening?

The information that you have access to is as follows:

party	arrival time	departure time
A	a_A	d_A
B	a_B	d_B
C	a_C	d_C
D	a_D	d_D
...	$a_{...}$	$d_{...}$

Challenge: Restaurant Problem

Solution

- sort the times a_i and b_i (as a single array)
- start a counter at zero
- set max_counter to zero
- for each element in the array of arrival/departure times
  - if it is an arrival, increment the counter
  - if it is a departure, decrement the counter
  - if counter > max_counter, set max_counter to counter
- max_counter stores the largest number of parties at the restaurant during the evening

Challenge: Restaurant Problem

Solution

- sort the times a_i and b_i (as a single array)
- start a counter at zero
- set max_counter to zero
- for each element in the array of arrival/departure times
  - if it is an arrival, increment the counter
  - if it is a departure, decrement the counter
  - if counter > max_counter, set max_counter to counter
- max_counter stores the largest number of parties at the restaurant during the evening

This is algorithm in the family of **sweep line** algorithms.

Challenge: Springbreak

Task

- You are planning an eventful springbreak, but your parents insist that you go back home to see them during the break.
- As a compromise, you are going back home, but you will do as many *fun things* as you can during that week.
- You have a calendar of fun events that are happening around you. The goal is to attend as many of them as possible, but some of them are overlapping in time.
- You are going to write an algorihtm that picks the largest number of events to attend given the start and end time for each event.

The information that you have access to is as follows:

event	start time	end time
1	s_1	e_1
2	s_2	e_2
...	...	...
N	s_N	e_N

Note: the duration of the events does not matter, you just want as many of them as possible.

Challenge: Springbreak

Solution 1 - sort by length ($e_i - s_i$)

- sort the events by lengths
- as long as there are more events to pick from
    - pick shortest one
    - throw out all events that conflict with it

Challenge: Springbreak

Solution 1 - sort by length ($e_i - s_i$)

- sort the events by lengths
- as long as there are more events to pick from
  - pick shortest one
  - throw out all events that conflict with it

Solution 2 - sort by start time (s_i)

- sort the events by their start time
- as long as there are more events to pick from
  - pick a next event (the one that starts as soon as possible)
  - throw out all the events that conflict with it

Challenge: Springbreak

Solution 1 - sort by length ($e_i - s_i$)

- sort the events by lengths
- as long as there are more events to pick from
  - pick shortest one
  - throw out all events that conflict with it

Solution 2 - sort by start time (s_i)

- sort the events by their start time
- as long as there are more events to pick from
  - pick a next event (the one that starts as soon as possible)
  - throw out all the events that conflict with it

Solution 3 - sort by end time (e_i)

- sort the events by their end time
- as long as there are more events to pick from
  - pick a next event (the one that ends as soon as possible)
  - throw out all the events that conflict with it

Challenge: Springbreak

Solution 1 - sort by length ($e_i - s_i$)

- sort the events by lengths
- as long as there are more events to pick from
  - pick shortest one
  - throw out all events that conflict with it

Solution 2 - sort by start time (s_i)

- sort the events by their start time
- as long as there are more events to pick from
  - pick a next event (the one that starts as soon as possible)
  - throw out all the events that conflict with it

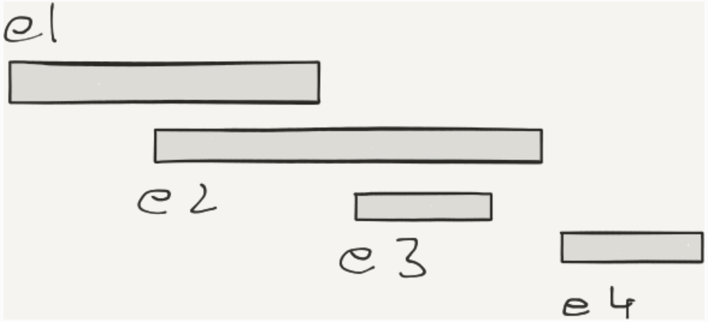
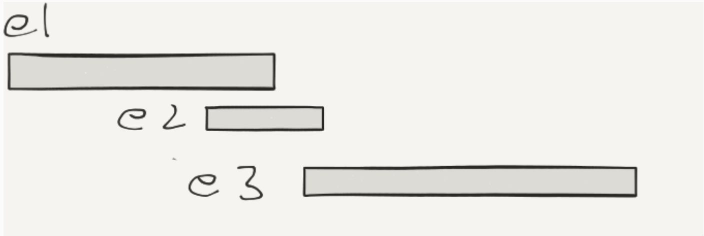
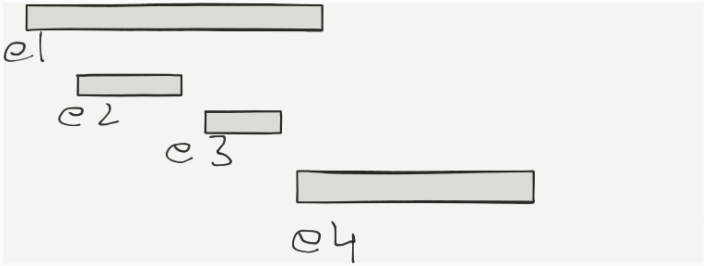
Solution 3 - sort by end time (e_i)

- sort the events by their end time
- as long as there are more events to pick from
  - pick a next event (the one that ends as soon as possible)
  - throw out all the events that conflict with it

Which of these solutions would result in the largest number of events?

Challenge: Springbreak

Consider these scenarios

scenario	sol 1	sol 2	sol 3	optimal
	2	2	2	2
	1	2	2	2
	3	1	3	3

Challenge: Springbreak

The solutions #3 is the optimal one.

Justification for optimality:
consider what happens when we pick an event that ends later than the one we picked

- it will either conflict with another event that is in the set of events for solution 3, or not
- if it does not conflict, than we have an equvalent optimal solution (i.e., same number of events)
- if it does conflict, we have to give up another event, and the new solution has fewer events (not optimal)

Picking an event that ends sooner, always gives a better or equivalent solution than picking an event that ends later.

Challenge: Springbreak

The solutions #3 is the optimal one.

Justification for optimality:
consider what happens when we pick an event that ends later than the one we picked

- it will either conflict with another event that is in the set of events for solution 3, or not
- if it does not conflict, than we have an equvalent optimal solution (i.e., same number of events)
- if it does conflict, we have to give up another event, and the new solution has fewer events (not optimal)

Picking an event that ends sooner, always gives a better or equivalent solution than picking an event that ends later.

🤓

So, the *computer scientist in you* spent the entire spring break analyzing different algorithms and never went to any events at all.

Searching

Searching Algorithms

- linear search
  - visits every element, $O(N)$
  - no assumptions about the data

- binary search
  - visits small fraction of elements, $O(\log N)$
  - data has to be sorted

Challenge: Count A's and B's

Taks

Given an array of `N` elements such that indexes `0..k` are filled with A's and indexes `k+1..N-1` are filled with B's, find the number of A's and B's in that array.

Example:

- input array `data = [A, A, A, B, B, B, B, B, B, B]`
- output: 3 A's and 7 B's

Challenge: Count A's and B's

Solution

Perform a modified binary search on `data` searching for an A that is followed by a B.

```
begin = 0
end = size of the array - 1

while begin <= end
      mid = (begin+end)/2
      if data[mid] == A
          if data[mid+1] == B            <=== look for a B
              found it so break
          else
              search in the first half
              end = mid-1
      else
          search in the second half
          begin = mid+1

countA = mid + 1
countB = n - mid - 1
```