Outline

1 Mid Eval
2 Sorting review
3 Bucketing
4 Hashing
5 One More Example
Outline

1. Mid Eval
2. Sorting review
3. Bucketing
4. Hashing
5. One More Example
Course eval

- Keep up, in general
Course eval

- Keep up, in general
- Too hard assignments
- Too many points needed
- Cooperation forbidden
Course eval

- Keep up, in general
- Too hard assignments
- Too many points needed
- Cooperation forbidden
- More examples!
Some-Algorithm$(A)$

1. $l \leftarrow \text{length}(A)$
2. \textbf{for} $i \leftarrow 1$ \textbf{to} $l$
3. \hspace{1em} \textbf{do} $i_m \leftarrow i$
4. \hspace{2em} \textbf{for} $j \leftarrow i + 1$ \textbf{to} $l$
5. \hspace{3em} \textbf{do if} $A[i_m] < A[j]$
6. \hspace{4em} \textbf{then} $i_m \leftarrow j$
7. \hspace{2em} $tmp \leftarrow A[i]$
8. \hspace{2em} $A[i] \leftarrow A[i_m]$
9. \hspace{2em} $A[i_m] \leftarrow tmp$
Outline

1. Mid Eval
2. Sorting review
3. Bucketing
4. Hashing
5. One More Example
Insertion Sort
Insertion Sort

- Simple
Insertion Sort

- Simple
- In place sorting algorithm
Insertion Sort

- Simple
- In place sorting algorithm
- $O(n^2)$ worst case behaviour
Merge Sort
Merge Sort

- Does not sort in place
Merge Sort

- Does not sort in place
- $\Theta(n \lg n)$ worst case behaviour
Heapsort

Heapsort Relies on the heap data structure
Max-Heapify is used to "bubble" up the values
Sorts in place
Θ(n lg n) worst case behaviour
Heapsort

- Relies on the heap data structure
Heapsort

- Relies on the heap data structure
- Max-Heapify is used to “bubble” up the values
Heapsort

- Relies on the heap data structure
- Max-Heapify is used to “bubble” up the values
- Sorts in place
Heapsort

- Relies on the heap data structure
- Max-Heapify is used to “bubble” up the values
- Sorts in place
- $\Theta(n \lg n)$ worst case behaviour
Quicksort

Very common in practice
Works with a pivot element and partitioning
$\Theta(n^2)$ worst case behaviour
$O(n \log n)$ average case behaviour
Quicksort

- Very common in practice
Quicksort

- Very common in practice
- Works with a pivot element and partitioning
Quicksort

- Very common in practice
- Works with a pivot element and partitioning
- $\Theta(n^2)$ worst case behaviour
Quicksort

- Very common in practice
- Works with a pivot element and partitioning
- $\Theta(n^2)$ worst case behaviour
- $O(n \log n)$ average case behaviour
Counting Sort

Assumes all keys are in the interval $[0, k)$

Works by counting the number of occurrences of each key

$\Theta(n + k)$ worst case
Counting Sort

- Assumes all keys are in the interval $[0, k)$
Counting Sort

- Assumes all keys are in the interval $[0, k)$
- Works by counting the number of occurrences of each key
Counting Sort

- Assumes all keys are in the interval $[0, k)$
- Works by counting the number of occurrences of each key
- $\Theta(n + k)$ worst case
Radix Sort

Sorts a "column" at a time using a stable sorting algorithm
$O(d(n+k))$ to sort $n$ $d$-digit numbers in the worst case
Radix Sort

- Sorts a “column” at a time using a stable sorting algorithm
Radix Sort

- Sorts a “column” at a time using a stable sorting algorithm
- $O(d(n + k))$ to sort $n$ $d$-digit numbers in the worst case
Bucket sort

Assumes input is uniformly distributed in \((0, 1)\).

Works by dividing input into buckets.

Expected running time \(\Theta(n)\).
**Bucket sort**

- Assumes input is uniformly distributed in $[0, 1)$
Bucket sort

- Assumes input is uniformly distributed in $[0, 1)$
- Works by dividing input into buckets
Bucket sort

- Assumes input is uniformly distributed in $[0, 1)$
- Works by dividing input into buckets
- Expected running time $\Theta(n)$
Example exam question

Problem

What is the maximum number of times during the execution of quicksort that the largest element can be moved, for an array of N elements? Explain your answer in no more than three lines.
Recall:

Partition \((A, p, r)\)

1. \(i \leftarrow A[r]\)
2. \(i \leftarrow p - 1\)
3. for \(j \leftarrow p\) to \(r - 1\)
   4. if \(A[j] \leq x\)
      5. then \(i \leftarrow i + 1\)
6. exchange \(A[i] \leftrightarrow A[j]\)
7. return \(i + 1\)

Quicksort \((A, p, r)\)

1. if \(p < r\)
   2. then \(q \leftarrow \text{Partition}(A, p, r)\)
   3. Quicksort \((A, p, q - 1)\)
   4. Quicksort \((A, q + 1, r)\)
Recall:

\text{Partition}(A, p, r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. for $j \leftarrow p$ to $r - 1$
4. do if $A[j] \leq x$
5. then $i \leftarrow i + 1$
6. exchange $A[i] \leftrightarrow A[j]$
7. return $i + 1$
Recall:

Partition($A, p, r$)

1. \( x \leftarrow A[r] \)
2. \( i \leftarrow p - 1 \)
3. for \( j \leftarrow p \) to \( r - 1 \)
4. \hspace{1cm} do if \( A[j] \leq x \)
5. \hspace{2cm} then \( i \leftarrow i + 1 \)
6. \hspace{1cm} exchange \( A[i] \leftrightarrow A[j] \)
7. return \( i + 1 \)

Quicksort($A, p, r$)

1. if \( p < r \)
2. then \( q \leftarrow \text{Partition}(A, p, r) \)
3. Quicksort($A, p, q - 1$)
4. Quicksort($A, q + 1, r$)
Outline

1. Mid Eval
2. Sorting review
3. Bucketing
4. Hashing
5. One More Example
Bucketing

Very useful technique
Hashing

- **Justification**: Hashing is good. Google uses hashing. So should you.
Hashing

- **Justification:** Hashing is good. Google uses hashing. So should you.
- **Principle:** Know where to put it.
Hashing

- **Justification**: Hashing is good. Google uses hashing. So should you.
- **Principle**: Know where to put it.
- **Performance**: Put equally many in each bucket.
Outline

1 Mid Eval
2 Sorting review
3 Bucketing
4 Hashing
5 One More Example
Problem

(15p)

Assume that we a set of four digit numbers that we want to store in a hash table:

1066 1789 1945 1600 1915 2005 1000

Consider two hash functions $\text{hashCode}_1(x) = x \mod 10$ and $\text{hashCode}_2(x) = \frac{x - (x \mod 1000)}{1000}$. Assume numbers arrive from left to right.

a) Draw the resulting hash table if we use $\text{hashCode}_1$ and linear probing to resolve collisions.

b) Draw the resulting hash table if we use $\text{hashCode}_2$ and chaining to resolve collisions.

c) With the additional knowledge that the input numbers are all years, which of the two hash functions would be the better choice for arbitrary input?
Problem

Consider the following four hash functions.

- \( h_1(k) = k \mod n \)
- \( h_2(k) = 2k \mod n \)
- \( h_3(k) = \lfloor k/10 \rfloor \mod n \)
- \( h_4(k) = k \cdot n \mod n \)

For each hash function, explain if it is a good or bad choice given that we use chaining and

1. all keys are even and \( n \) is divisible by 4
2. \( n > 1000 \) and all keys are smaller than 5000
3. \( n \) is odd

Justify your answer in one sentence.