CS 4102: Algorithms
Lecture 9: Quicksort and Quickselect

David Wu
Fall 2020
Homework

Hashing Exercise 1 due September 28

- Written (use LaTeX!)
- Randomized algorithms
- Hash functions
Divide and Conquer Quiz

Divide and Conquer Quiz
- Due today, 11:59pm
- Written assignment (questions similar to written exercises)
- Open book/note, but no collaboration/sharing of documents

Divide and Conquer Quiz Re-attempt (Optional)
- Released Monday, September 28, due Tuesday, September 29

Only grade on last attempt on quiz counts
Hash Tables

Universe of possible elements

Hash function $H$
Maps an element to bucket index
Hash Tables

**Chaining:** represent each bucket as a linked list of values

**Insertion query:** hash the element $x$ and add element to bucket

**Deletion query:** hash the element $x$ and remove from bucket (if present)

**Lookup query:** hash the element $x$ and check if it is in the bucket

**Running time?**

$O(1 + X)$ where $X$ is the number of elements in the bucket

$O(1)$ to evaluate the hash, $O(X)$ to traverse bucket

Suppose there are $n$ elements and $m$ buckets in hash table

- **Best case:** Elements evenly distributed: $O(1 + n/m)$
- **Worst case:** Elements hash to one bucket: $O(n)$
Choosing a Good Hash Function

“Good” hash function: evenly distribute elements

“Bad” hash function: concentrates elements in few buckets
Choosing a Good Hash Function

Does there exist a good hash function?

Let $U$ be the universe of elements, $n$ be the number of elements, and $m$ be the number of buckets.

**Bad news:** For every hash function $H : U \rightarrow \{1, \ldots, m\}$, if $|U| > m(n - 1)$, there exists $x_1, \ldots, x_n$ such that $H(x_1) = H(x_2) = \cdots = H(x_n)$.

**Implication:** When universe is large, there is always a bad set of inputs where hash table costs are $O(n)$.

Avoid worst case behavior: choose a hash function at random.
Let $\mathcal{H}$ be a collection of hash functions from $U \rightarrow \{1, ..., m\}$

The hash function $\mathcal{H}$ is universal if for all $k \neq \ell \in U$,

$$\Pr_{h \leftarrow \mathcal{H}}[h(k) = h(\ell)] = \frac{1}{m}$$

Number of hash functions where $h(k) = h(\ell)$ is $|\mathcal{H}|/m$ for all $k \neq \ell \in U$

**Hash table construction:** when the table is initialized, sample a random hash function $h \leftarrow \mathcal{H}$
Expected Cost with Universal Hashing

Suppose there are $n$ distinct keys in a hash table $T$ with $m$ buckets.

Suppose that the hash function $h$ is sampled from a universal hash family $\mathcal{H}$.

Recall: cost of operations is $O(1 + X)$ where $X$ is the bucket size.

For a key $k \in U$, let $Y_k$ be the number of keys $\ell \in T$ where $h(\ell) = h(k)$ and $\ell \neq k$.

For keys $k, \ell \in U$, let $Y_{k\ell}$ be an indicator random variable for the event $h(k) = h(\ell)$.

- $\mathcal{H}$ is universal: $\Pr[h(k) = h(\ell)] = 1/m$.
- $\mathbb{E}[Y_{k\ell}] = \Pr[h(k) = h(\ell)] = 1/m$.
- $\mathbb{E}[Y_k] = \mathbb{E}\left[\sum_{\ell \in T, \ell \neq k} Y_{k\ell}\right] = \sum_{\ell \in T, \ell \neq k} \mathbb{E}[Y_{k\ell}] = \sum_{\ell \in T} \frac{1}{m}$.
Expected Cost with Universal Hashing

\(Y_k\): number of keys \(\ell \in T\) where \(h(\ell) = h(k)\) and \(\ell \neq k\)

\[
\mathbb{E}[Y_k] = \sum_{\substack{\ell \in T \\ell \neq k}} \frac{1}{m}
\]

**Case 1:** We search for a key \(k \notin T\)
- Expected cost \(c_k\) is size of bucket containing \(h(k)\)
- \(\mathbb{E}[c_k] = \mathbb{E}[Y_k] = n/m = \alpha\)

**Case 2:** We search for a key \(k \in T\)
- Expected cost \(c_k\) is size of bucket containing \(h(k)\)
- \(\mathbb{E}[c_k] = \mathbb{E}[Y_k] + 1 = \frac{n-1}{m} + 1 = 1 + \alpha - \frac{1}{m} < 1 + \alpha\)

Recall: \(\alpha = n/m\) is the load factor

Expected lookup cost: \(O(1 + \alpha)\)

**Conclusion:** setting \(m = O(n)\) yields expected constant time for all operations
Quick sort:

- Divide and conquer algorithm
- $O(n \log n)$ run time (on expectation); can be guaranteed using “median-of-median” approach

Unlike merge sort:

- **Divide** step is the “expensive” step
- Typically faster than merge sort (often is the basis of sorting algorithms in standard library implementations)
Quicksort

**General idea:** choose a pivot element, recursively sort two sublists around that element

**Divide:** select pivot element $p$, $\text{Partition}(p)$

**Conquer:** recursively sort left and right sublists

**Combine:** nothing!
Partition Procedure (Divide Step)

**Input:** an unordered list, a pivot $p$

![List](image)

**Goal:** All elements $< p$ on left, all $\geq p$ on right

![Sorted List](image)
Partition Procedure

Initialize two pointers **Begin** and **End**
Partition Procedure

If Begin value < \( p \), move Begin right
Else swap Begin value with End value, move End Left
Stop when Begin = End

8 5 7 3 12 10 1 2 4 9 6 11

Swap!
If \( \text{Begin value} < p \), move \( \text{Begin right} \)
Else swap \( \text{Begin value} \) with \( \text{End value} \), move \( \text{End Left} \)
Stop when \( \text{Begin} = \text{End} \)

Partition Procedure
If Begin value < p, move Begin right
Else swap Begin value with End value, move End Left
Stop when Begin = End

Partition Procedure

1. 8  5  7  3  6  10  1  2  4  9  11  12
   Swap!

2. 8  5  7  3  6  9  1  2  4  10  11  12
   Swap!

3. 8  5  7  3  6  4  1  2  9  10  11  12
   Swap!

4. 8  5  7  3  6  4  1  2  9  10  11  12
If Begin value < \( p \), move Begin right
Else swap Begin value with End value, move End Left
Stop when Begin = End

Remaining item: where do we place the pivot?
If Begin value $< p$, move Begin right
Else swap Begin value with End value, move End Left
Stop when Begin = End

Case 1: meet at element $< p$
Swap $p$ with pointer position
Partition Procedure

If `Begin` value < \( p \), move `Begin` right
Else swap `Begin` value with `End` value, move `End` Left
Stop when `Begin` = `End`

**Case 2:** meet at element > \( p \)
Swap \( p \) with value to the left
Partition Procedure Summary

1. Choose the pivot $p$ to be the first element of the list
2. Initialize two pointers Begin (just after $p$), and End (at end of list)
3. While Begin < End:
   • If value of Begin < $p$, advance Begin to the right
   • Otherwise, swap value of Begin value with value of End value, and advance End to the left
4. If pointers meet at element $< p$: swap $p$ with pointer position
5. Otherwise, if pointers meet at element $> p$: swap $p$ with value to the left

Run time? $\Theta(n)$
Conquer Step

2  5  7  3  6  4  1  8  9  10  11  12

All elements $< p$  All elements $> p$

Exactly where it belongs!

Recursively sort **Left** and **Right** sublists
Quicksort Run Time (Optimistic)

If the **pivot** is the **median**:

Then we divide in half each time

\[ T(n) = 2T(n/2) + n = \Theta(n \log n) \]
If the pivot is the extreme (min/max):

\[
\begin{array}{ccccccccccccc}
1 & 5 & 2 & 3 & 6 & 4 & 7 & 8 & 10 & 9 & 11 & 12
\end{array}
\]

Then we shorten by 1 each time

\[
T(n) = T(n - 1) + n
\]

\[
= n + (n - 1) + \cdots + 2 + 1
\]

\[
= \frac{n(n + 1)}{2} = \Theta(n^2)
\]
First element always yields unbalanced pivot

Then we shorten by 1 each time

\[ T(n) = \Theta(n^2) \]
How to Choose the Pivot?

Good choice: $\Theta(n \log n)$

Bad choice: $\Theta(n^2)$
What makes a good pivot?
  • Roughly even split between left and right
  • Ideally: median

Can we find median in linear time?
  • Yes! Quickselect algorithm
Quickselect Algorithm

Algorithm to compute the $i^{th}$ order statistic

- $i^{th}$ smallest element in the list
- 1$^{st}$ order statistic: minimum
- $n^{th}$ order statistic: maximum
- $(n/2)^{th}$ order statistic: median
Quickselect Algorithm

Finds $i^{th}$ order statistic

**General idea:** choose a pivot element, partition around the pivot, and recurse on sublist containing index $i$

**Divide:** select pivot element $p$, $\text{Partition}(p)$

**Conquer:**
- if $i =$ index of $p$, then we are done and return $p$
- if $i <$ index of $p$ recurse left. Otherwise, recurse right

**Combine:** Nothing!
Partition Procedure (Divide Step)

**Input:** an unordered list, a pivot $p$

| 8 | 5 | 7 | 3 | 12 | 10 | 1 | 2 | 4 | 9 | 6 | 11 |

**Goal:** All elements $< p$ on left, all $\geq p$ on right

| 5 | 7 | 3 | 1 | 2 | 4 | 6 | 8 | 12 | 10 | 9 | 11 |
Conquer Step

All elements $< p$

All elements $> p$

Correct position of $p$

Recurse on sublist that contains index $i$

(subtract index of the pivot to $i$ if recursing right)
Quickselect Run Time (Optimistic)

If the **pivot** is the **median**:

Then we divide in half each time

\[ T(n) = T(n/2) + n = \Theta(n) \]
Quickselect Run Time (Worst-Case)

If the **pivot** is the **extreme** (min/max):

\[
T(n) = T(n - 1) + n = \Theta(n^2)
\]

Then we shorten by 1 each time
How to Choose the Pivot?

Good choice: $\Theta(n)$

Bad choice: $\Theta(n^2)$
What makes a good pivot?

- Roughly even split between left and right
- Ideally: median

But this is the problem that Quickselect is supposed to solve!

**What’s next:** an algorithm for choosing a “decent” pivot (median of medians)