Analysis of Algorithms

Sorting

Andres Mendez-Vazquez

September 23, 2020

Outline

1. Sorting $O(n \log n)$
   - Divide and Conquer
   - HeapSort
   - QuickSort
2. Linear Sorting
   - Counting Sort
   - Radix Sort
   - Bucket Sort
3. Median Statistics
   - Selection in Expected Linear Time
   - Worst Case Median Statistics

Merge Problem

Problem

- Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

You could use the following strategy

- Merge the first two arrays with extra memory, then merge in the third, then merge in the fourth, and so on.
  - What is the time complexity of this algorithm, in terms of k and n?

Then

- Give a more efficient solution to this problem, using divide-and-conquer.

Merge Problem

Problem

- Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

You could use the following strategy

- Merge the first two arrays with extra memory, then merge in the third, then merge in the fourth, and so on.
  - What is the time complexity of this algorithm, in terms of k and n?

Then

- Give a more efficient solution to this problem, using divide-and-conquer.

Merge Problem

Problem

- Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

You could use the following strategy

- Merge the first two arrays with extra memory, then merge in the third, then merge in the fourth, and so on.
  - What is the time complexity of this algorithm, in terms of k and n?

Then

- Give a more efficient solution to this problem, using divide-and-conquer.

Outline

1. Sorting $O(n \log n)$
   - Divide and Conquer
   - HeapSort
   - QuickSort
2. Linear Sorting
   - Counting Sort
   - Radix Sort
   - Bucket Sort
3. Median Statistics
   - Selection in Expected Linear Time
   - Worst Case Median Statistics

Worst Case

Problem

- Show that the worst-case running time of Max-Heapify on a heap of size n is $\Omega(\log n)$.
  - Hint: For a heap with n nodes, give node values that cause Max-Heapify to be called recursively at every node on a simple path from the root down to a leaf.

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

1. Build-Max-Heap(A)
2. for $i = length[A]$ downto 2
3. exchange $A[1]$ with $A[i]$
4. $heap - size[A] = heap - size[A] - 1$
5. **Max-Heapify**$(A, 1)$

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

1. Build-Max-Heap(A)
2. for $i = length[A]$ downto 2
3.   exchange $A[1]$ with $A[i]$
4. $heap - size[A] = heap - size[A] - 1$
5.   **Max-Heapify**$(A, 1)$

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

1. Build-Max-Heap(A)
2. for $i = length[A]$ downto 2
3. exchange $A[1]$ with $A[i]$
4. $heap - size[A] = heap - size[A] - 1$
5. **Max-Heapify**$(A, 1)$

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

1. Build-Max-Heap(A)
2. for $i = length[A]$ downto 2
3. exchange $A[1]$ with $A[i]$
4. $heap-size[A] = heap-size[A] - 1$
5. **Max-Heapify**$(A, 1)$

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

1. Build-Max-Heap(A)
2. for $i = length[A]$ downto 2
3. exchange $A[1]$ with $A[i]$
4. $heap - size[A] = heap - size[A] - 1$
5. Max-Heapify$(A, 1)$

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

Build-Max-Heap(A)
for $i = length[A]$ downto 2
    exchange $A[1]$ with $A[i]$
    $heap - size[A] = heap - size[A] - 1$
    Max-Heapify$(A, 1)$

Figure: Heapsort

Heap Sort: Using Max-Heapify

Heapsort Algorithm

Heapsort(A)

1. Build-Max-Heap(A)
2. for $i = length[A]$ downto 2
3. exchange $A[1]$ with $A[i]$
4. $heap - size[A] = heap - size[A] - 1$
5. **Max-Heapify**$(A, 1)$

Figure: Heapsort

Loop Invariance

Argue the correctness of HeapSort using the following loop invariant

- At the start of each iteration of the for loop of lines 2–5, the subarray $A[1, ..., i]$ is a max-heap containing the i smallest elements of $A[1, ..., n]$ sorted, and also the subarray $A[i+1, ..., n]$ contains the $n - i$ largest elements of $A[1, ..., n]$, sorted.

Outline

1. Sorting $O(n \log n)$
   - Divide and Conquer
   - HeapSort
   - QuickSort
2. Linear Sorting
   - Counting Sort
   - Radix Sort
   - Bucket Sort
3. Median Statistics
   - Selection in Expected Linear Time
   - Worst Case Median Statistics

Problems

Reversing Order

- How would you modify QuickSort to sort into nonincreasing order?

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $q = Partition\ (A, p, r)$
3. **Quicksort**$(A, p, q - 1)$
4. **Quicksort**$(A, q + 1, r)$

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$ Quicksort$(A, p, q-1)$
4. $\quad$ Quicksort$(A, q+1, r)$

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$ **Quicksort**$(A, p, q-1)$
4. $\quad$ **Quicksort**$(A, q+1, r)$

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$ **Quicksort**$(A, p, q - 1)$
4. $\quad$ **Quicksort**$(A, q + 1, r)$

Quicksort Algorithm

Quicksort Algorithm

Quicksort(A, p, r)

1. if $p < r$
2. $\quad q = Partition\,(A, p, r)$
3. $\quad$ **Quicksort**$(A, p, q-1)$
4. $\quad$ **Quicksort**$(A, q+1, r)$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4.   if $A[j] \leq x$
5. $i = i + 1$
6.   exchange $A[i]$ with $A[j]$
7. exchange $A[i + 1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i + 1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4.   if $A[j] \leq x$
5. $i = i + 1$
6.   exchange $A[i]$ with $A[j]$
7. exchange $A[i + 1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4.   if $A[j] \leq x$
5. $i = i + 1$
6.   exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4.   if $A[j] \leq x$
5. $i = i + 1$
6.   exchange $A[i]$ with $A[j]$
7. exchange $A[i + 1]$ with $A[r]$
8. return $i + 1$

Partition Algorithm

Quicksort Partition

Partition(A, p, r)

1. $x = A[r]$
2. $i = p - 1$
3. for $j = p$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i+1]$ with $A[r]$
8. return $i + 1$

Another Problem

Equal Values

- What value of q does **Partition** return when all elements in the array $A[p, ..., r]$ have the same value?

Then

- Modify **Partition** so that $q = \left\lfloor \frac{p+r}{2} \right\rfloor$ when all elements in the array $A[p, ..., r]$ have the same value.

Another Problem

Equal Values

- What value of q does **Partition** return when all elements in the array $A[p, ..., r]$ have the same value?

Then

- Modify **Partition** so that $q = \left\lfloor \frac{p+r}{2} \right\rfloor$ when all elements in the array $A[p, ..., r]$ have the same value.

Outline

1. Sorting $O(n \log n)$
   - Divide and Conquer
   - HeapSort
   - QuickSort
2. Linear Sorting
   - Counting Sort
   - Radix Sort
   - Bucket Sort
3. Median Statistics
   - Selection in Expected Linear Time
   - Worst Case Median Statistics

Preprocessing

Describe an algorithm that, given n integers in the range 0 to k

- Preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ in $O(1)$
  - You have $O(n + k)$ preprocessing time.

Outline

Induction

Problem

1. Use induction to prove that radix sort works.
2. Where does your proof need the assumption that the intermediate sort is stable?

How to use logs

Problem

- Show how to sort n integers in the range 0 to $n^3 \$ 1$ in $O(n)$ time.

Outline

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B[0..n-1]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4. make $B[i]$ an empty list
5. for $i = 0$ to n
6. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
7. for $i = 0$ to $n-1$
8. sort list $B[i]$ with insertion sort
9. concatenate the list $B[0], B[1], \ldots, B[n-1]$ together in order

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B\left[0..n-1\right]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4. make $B\left[i\right]$ an empty list
5. for $i = 0$ to n
6. insert $A\left[i\right]$ into list $B\left[\lfloor nA\left[i\right]\rfloor\right]$
7. for $i = 0$ to $n-1$
8. sort list $B\left[i\right]$ with insertion sort
9. concatenate the list $B\left[0\right]$, $B\left[1\right]$, ... , $B\left[n-1\right]$ together in order

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B[0..n-1]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4. make $B[i]$ an empty list
5. for $i = 0$ to n
6. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
7. for $i = 0$ to $n-1$
8. sort list $B[i]$ with insertion sort
9. concatenate the list $B[0], B[1], \ldots, B[n-1]$ together in order

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B\left[0..n-1\right]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4.   make $B\left[i\right]$ an empty list
5. for $i = 0$ to n
6.   insert $A\left[i\right]$ into list $B\left[\lfloor nA\left[i\right]\rfloor\right]$
7. for $i = 0$ to $n-1$
8.   sort list $B\left[i\right]$ with insertion sort
9. concatenate the list $B\left[0\right], B\left[1\right], \ldots, B\left[n-1\right]$ together in order

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B\left[0..n-1\right]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4.   make $B\left[i\right]$ an empty list
5. for $i = 0$ to n
6.   insert $A\left[i\right]$ into list $B\left[\lfloor nA\left[i\right]\rfloor\right]$
7. for $i = 0$ to $n-1$
8.   sort list $B\left[i\right]$ with insertion sort
9. concatenate the list $B\left[0\right], B\left[1\right], \ldots, B\left[n-1\right]$ together in order

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B[0..n-1]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4.   make $B[i]$ an empty list
5. for $i = 0$ to n
6.   insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
7. for $i = 0$ to $n-1$
8.   sort list $B[i]$ with insertion sort
9. concatenate the list $B[0], B[1], \ldots, B[n-1]$ together in order

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B\left[0..n-1\right]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4.   make $B\left[i\right]$ an empty list
5. for $i = 0$ to n
6.   insert $A\left[i\right]$ into list $B\left[\lfloor nA\left[i\right]\rfloor\right]$
7. for $i = 0$ to $n-1$
8.   sort list $B\left[i\right]$ with insertion sort
9. concatenate the list $B\left[0\right]$, $B\left[1\right]$, ... , $B\left[n-1\right]$ together in order

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B\left[0..n-1\right]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4. make $B\left[i\right]$ an empty list
5. for $i = 0$ to n
6. insert $A\left[i\right]$ into list $B\left[\lfloor nA\left[i\right]\rfloor\right]$
7. for $i = 0$ to $n-1$
8. sort list $B\left[i\right]$ with insertion sort
9. concatenate the list $B\left[0\right]$, $B\left[1\right]$, ... , $B\left[n-1\right]$ together in order

Bucket Sort Algorithm

Algorithm assuming

Buket-Sort(A)

1. let $B\left[0..n-1\right]$ be a new array
2. $n = A.length$
3. for $i = 0$ to $n-1$
4. make $B\left[i\right]$ an empty list
5. for $i = 0$ to n
6. insert $A\left[i\right]$ into list $B\left[\lfloor nA\left[i\right]\rfloor\right]$
7. for $i = 0$ to $n-1$
8. sort list $B\left[i\right]$ with insertion sort
9. concatenate the list $B\left[0\right]$, $B\left[1\right]$, ... , $B\left[n-1\right]$ together in order

Questions

We have

- Explain why the worst-case running time for bucket sort is $O(n)$

Now

- What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \log n)$

Questions

We have

- Explain why the worst-case running time for bucket sort is $O\left(n\right)$

Now

- What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O\left(n\log n\right)$

Outline

We have the following

$X_k = I\{$the subarray A[p .. q] has exactly $k$elements$\}$ with $E[X_k] = \frac{1}{n}$ (Assuming that the elements are distinct)

$$
\begin{aligned}
T(n) &\leq \sum_{k=1}^{n} X_k \times \left(T(\max(k-1, n-k)) + O(n)\right) \\
&= \sum_{k=1}^{n} X_k \times \left(T(\max(k-1, n-k)\right) + O(n)
\end{aligned}
$$

Thus

- Argue that the indicator random variable X_k and the value $T(\max(k-1, n-k)$ are independent.

We have the following

$X_k = I\{$the subarray A[p .. q] has exactly $k$elements$\}$ with $E[X_k] = \frac{1}{n}$ (Assuming that the elements are distinct)

$$
\begin{aligned}
T(n) &\leq \sum_{k=1}^{n} X_k \times (T(\max(k-1, n-k)) + O(n)) \\
&= \sum_{k=1}^{n} X_k \times (T(\max(k-1, n-k)) + O(n)
\end{aligned}
$$

Thus

- Argue that the indicator random variable X_k and the value $T(\max(k-1, n-k)$ are independent.

Outline

Imagine

Black-Box

- Suppose that you have a "black-box" worst-case linear-time median subroutine.
    - Give a simple, linear-time algorithm that solves the selection problem for an arbi- trary order statistic