CS170: Discrete Methods in Computer Science
Summer 2023
Sorting

Instructor: Shaddin Dughmi[1]

[1] These slides adapt some content from similar slides by Aaron Cote.

Sorting

- In this lecture, we will examine the problem of sorting an array.
- This will exercise what we learned about proofs (especially induction) and runtime.
- Good warmup for 270.

Sorting

- In this lecture, we will examine the problem of sorting an array.
- This will exercise what we learned about proofs (especially induction) and runtime.
- Good warmup for 270.

Sorting

- Input: An array of n numbers, in arbitrary order
  - $a = [a_0, \ldots, a_{n-1}]$
- Output: Array with same n numbers, ordered from small to large
  - If the same number appears multiple times in the input, it must appear the same number of times in the output

Sorting

- In this lecture, we will examine the problem of sorting an array.
- This will exercise what we learned about proofs (especially induction) and runtime.
- Good warmup for 270.

Sorting

- Input: An array of n numbers, in arbitrary order
  - $a = [a_0, \ldots, a_{n-1}]$
- Output: Array with same n numbers, ordered from small to large
  - If the same number appears multiple times in the input, it must appear the same number of times in the output

Computational Model

We need to be clear about how to count runtime. In addition to usual, we consider the following to be basic operations taking constant time:

- Comparison of two numbers
- Reading / Writing from array, given index

Outline

Bubble Sort at a High Level

- Compare 1st and 2nd, swap if out of order
- Compare 2nd and 3rd, swap if out of order
- All the way to nth

Bubble Sort at a High Level

- Compare 1st and 2nd, swap if out of order
- Compare 2nd and 3rd, swap if out of order
- All the way to nth

At this point, we know the largest is at position n, so we repeat the above up to position $n-1$, then up to position $n-2$, etc

Bubble Sort at a High Level

- Compare 1st and 2nd, swap if out of order
- Compare 2nd and 3rd, swap if out of order
- All the way to nth

At this point, we know the largest is at position n, so we repeat the above up to position $n-1$, then up to position $n-2$, etc

Lets work through this example: [7, 9, 5, 9, 3]

Bubble Sort in Pseudocode

- For i from $n-1$ to 1
  - For j from 0 to $i-1$
    - If $a[j] > a[j+1]$ then swap(a,j, $j+1$)

Bubble Sort in Pseudocode

- For i from $n-1$ to 1
  - For j from 0 to $i-1$
    - If $a[j] > a[j+1]$ then swap(a,j, $j+1$)

The function swap(a,j,$j+1$) just reads $a[j]$ and $a[j+1]$ into registers and then copies $a[j]$ into position $j+1$ and $a[j+1]$ into position j. Obviously constant time.

Runtime Analysis

- For i from $n-1$ to 1
  - For j from 0 to $i-1$
    - If $a[j] > a[j+1]$ then swap(a,j, $j+1$)

Runtime Analysis

- $n-1 = O(n)$ iterations of outer loop
- $(n-1)+(n-2)+\ldots+1 = \sum_{i=1}^{n-1} = O(n^2)$ iterations of inner loop.
- Innermost statement takes time $O(1)$, executed $O(n^2)$ times
- Total: $O(n^2)$

Correctness

- For i from $n-1$ to 1
  - For j from 0 to $i-1$
    - If $a[j] > a[j+1]$ then swap(a,j, $j+1$)

We use induction on number of iterations of the outer loop to prove

Loop invariant

At the start of the kth iteration, the largest $k-1$ elements are in the last $k-1$ positions, in sorted order.

- kth iteration is when $i = n-k$.

Correctness

- For i from $n-1$ to 1
  - For j from 0 to $i-1$
    - If $a[j] > a[j+1]$ then swap(a,j, $j+1$)

We use induction on number of iterations of the outer loop to prove

Loop invariant

At the start of the kth iteration, the largest $k-1$ elements are in the last $k-1$ positions, in sorted order.

- kth iteration is when $i = n-k$.

Proof

- Base case: At start of 1st iter, largest 0 elts are in last 0 positions.
- Inductive Hypothesis: Loop invariant true for k
- Induction step: Prove Loop invariant for $k+1$. During iteration k, last $k-1$ elements (largest) don't move. kth largest element will be bubbled up to index $n-k$. So at start of $k+1$ iteration largest k elements are in the last k positions in sorted order.

Outline

Selection Sort at a High Level

- Scan array to find smallest element, swap into first position
- Scan array from 2nd to last element to find smallest, swap into in 2nd position
- Scan array from 3rd to last element to find smallest, swap into in 3rd position
- ...
- Until sorted

Selection Sort at a High Level

- Scan array to find smallest element, swap into first position
- Scan array from 2nd to last element to find smallest, swap into in 2nd position
- Scan array from 3rd to last element to find smallest, swap into in 3rd position
- ...
- Until sorted

Lets work through this example: [7, 9, 5, 9, 3]

Selection Sort in Pseudocode

- For i from 0 to $n-1$
  - $small = i$;
  - For j from $i+1$ to $n-1$
    - If $a[j] < a[small]$ then $small = j$
  - swap(a,i, $small$)

Runtime Analysis

- For i from 0 to $n-1$
  - $small = i$;
  - For j from $i+1$ to $n-1$
    - If $a[j] < a[small]$ then $small = j$
  - swap(a,i, $small$)

Runtime Analysis

- n iterations of outer loop
- $(n-1)+(n-2)+\ldots+1 = O(n^2)$ iterations of inner loop
- Otherwise, each statement takes constant time per execution
- Total: $O(n^2)$

Correctness

- For i from 0 to $n-1$
  - $small = i$;
  - For j from $i+1$ to $n-1$
    - If $a[j] < a[small]$ then $small = j$
  - swap(a,i, $small$)

Loop invariant

At the start of outer iteration with $i = k$, the smallest k elements are in the first k positions, in sorted order.

Correctness

- For i from 0 to $n-1$
  - $small = i$;
  - For j from $i+1$ to $n-1$
    - If $a[j] < a[small]$ then $small = j$
  - swap(a,i, $small$)

Loop invariant

At the start of outer iteration with $i = k$, the smallest k elements are in the first k positions, in sorted order.

Proof

- Base case: Smallest 0 elts are in first 0 positions at beginning.
- Inductive Hypothesis: Loop invariant true for k
- Induction step: Prove Loop invariant for $k+1$. During iteration with $i = k$, first k elements (smallest) don't move. $k+1$st smallest element will be swapped into position k (the $k+1$st position in the array). So at start of iteration with $i = k+1$, smallest $k+1$ elements are in the first $k+1$ positions in sorted order.

Outline

Insertion Sort at a High Level

- Sort the first element of the array (i.e., do nothing)
- Insert the second element of array so that first two elements are sorted
- Insert the third element so first three elements are sorted
- . . .
- Insert the last element so all elements are sorted.

Insertion Sort at a High Level

- Sort the first element of the array (i.e., do nothing)
- Insert the second element of array so that first two elements are sorted
- Insert the third element so first three elements are sorted
- ...
- Insert the last element so all elements are sorted.

Lets work through this example: [7, 9, 5, 9, 3]

Insertion Sort in Pseudocode

- For i from 1 to $n-1$
  - $j = i$
  - While ($j > 0$ and $a[j] < a[j-1]$)
    - swap(a,j,$j-1$)
    - $j = j - 1$

Runtime Analysis

- For i from 1 to $n-1$
  - $j = i$
  - While ($j > 0$ and $a[j] < a[j-1]$)
    - swap(a,j,$j-1$)
    - $j = j - 1$

Runtime Analysis

- $n - 1 = O(n)$ iterations of outer loop
- $1 + 2 \ldots + (n-1) = O(n^2)$ iterations of inner loop
- Otherwise, each statement takes constant time per execution
- Total: $O(n^2)$

Correctness

- For i from 1 to $n-1$
  - $j = i$
  - While ($j > 0$ and $a[j] < a[j-1]$)
    - swap(a,j,$j-1$)
    - $j = j - 1$

Loop invariant

At the start of outer iteration with $i = k$, the first k elements of the array are in sorted order.

Correctness

- For i from 1 to $n-1$
  - $j = i$
  - While ($j > 0$ and $a[j] < a[j-1]$)
    - swap(a,j,$j-1$)
    - $j = j - 1$

Loop invariant

At the start of outer iteration with $i = k$, the first k elements of the array are in sorted order.

Proof

- Base case: The first element is in sorted order
- Inductive Hypothesis: Loop invariant true for k
- Induction step: Prove Loop invariant for $k+1$. During iteration with $i = k$, first k elements don't change their relative order. The $k+1$st element will be inserted (bubbled down) in its proper place between them. So at start of iteration with $i = k+1$, first $k+1$ elements are in sorted order.

Outline

Merge Sort at a High Level

- If your array has 1 element, you're done
- Otherwise, resursively sort left half and right half
- Merge the left and right half to produce the entire sorted array
  - Smallest element overall must be smallest on left or on right
  - Most that to final array
  - Repeat

Merge Sort at a High Level

- If your array has 1 element, you're done
- Otherwise, resursively sort left half and right half
- Merge the left and right half to produce the entire sorted array
  - Smallest element overall must be smallest on left or on right
  - Most that to final array
  - Repeat

Lets work through this example: [1, 5, 3, 4, 2, 6]

Merge Sort in Pseudocode

$Mergsort(a, L, R)$:

- If $L = R$ return.
- Let m be the middle between L and R (what should exact equation for m be?)
- Mergesort(a, L,m)
- Mergesort(a,$m + 1$,R)
- Merge(a,L,m,R)

Merge Sort in Pseudocode

$Merge(a, L, m, R)$:

- Create temporary array b (how long?)
- $i = L$, $j = m + 1$, $k = 0$
- While $i \leq m$ and $j \leq r$
  - Copy the smaller of $a[i]$ and $a[j]$ to $b[k]$, incrementing the corresponding index (i or j), and incrementing k.
- Copy the remaining (uncopied) elements to b in order
- Copy b back to $a[L, \ldots, R]$.

Correctness

Lemma

Let a_ℓ denote the subarray $a[L, \ldots, m]$ and a_r denote the subarray $a[m+1, \ldots, R]$. When a_ℓ and a_r are sorted, merge sorts $a[L \ldots, R]$.

Correctness

Lemma

Let a_ℓ denote the subarray $a[L, \ldots, m]$ and a_r denote the subarray $a[m+1, \ldots, R]$. When a_ℓ and a_r are sorted, merge sorts $a[L \ldots, R]$.

Proof

- It suffices to prove that after the iteration of the while loop with $k = t$, $b[0, \ldots, t]$ contains the t smallest elements in $a[L, \ldots, R]$ in order.
  - The rest, after the while loop, is obvious since we copy remaining elements in order.
- We do this by induction on t.

Correctness

Lemma

Let a_ℓ denote the subarray $a[L, \ldots, m]$ and a_r denote the subarray $a[m+1, \ldots, R]$. When a_ℓ and a_r are sorted, merge sorts $a[L \ldots, R]$.

Proof

- Base case $t = 0$: We copy the smaller of $a[L]$ and $a[m+1]$ to $b[0]$. Since a_ℓ and a_r are sorted, this is smallest overall in $a[L, \ldots, R]$.
- Inductive hypothesis: After iteration of the while loop with $k = t$, $b[0, \ldots, t]$ contains the t smallest elements in $a[L, \ldots, R]$ in order.
- Inductive step:
  - Consider the iteration with $k = t + 1$.
  - $a[i]$ is the smallest element of a_ℓ that has not been copied to b, and $a[j]$ is the smallest element of a_r that has not been copied to b.
  - We pick smaller of these to copy into $b[k]$. This is the next smallest.
  - Therefore, $b[0, \ldots, t+1]$ now contains the $t+1$ smallest elements in order.

Correctness

Theorem

Mergesort correctly sorts the subarray $a' = a[L, \ldots, R]$.

Proof

- We induct on the length n of a'.
- Base case $n = 1$: Here $L = R$, and the algorithm returns.
- Induction hypothesis: Mergesort correctly sorts subarrays of length at most n.
- Induction step:
  - Consider subarray a' of length $n + 1$.
  - Mergesort splits it into two parts $a'_\ell = a[L, \ldots, m]$ and $a'_r = a[m + 1, \ldots, R]$ of length no more than n (in fact, roughly $\frac{n+1}{2}$), and recurses on each part.
  - By the induction hypothesis, the recursive calls correctly sort a'_ℓ and a'_r.
  - By our Lemma, Merge correctly sorts a' given the two sorted parts a'_ℓ and a'_r

Runtime Analysis: Merge

Lemma

The Merge operation runs in linear time.

In more detail: When given subarrays $a_\ell = a[L, \ldots, m]$ and $a_r = a[m+1, \ldots, R]$, with total length $n = R - L + 1$, runs in time $O(n)$.

Runtime Analysis: Merge

Lemma

The Merge operation runs in linear time.

In more detail: When given subarrays $a_\ell = a[L, \ldots, m]$ and $a_r = a[m+1, \ldots, R]$, with total length $n = R - L + 1$, runs in time $O(n)$.

Proof

- Creating b takes linear time
- While loop has $O(n)$ iterations, since it increments one of i or j each iteration, and this can happen at most $m - L + R - m = O(n)$ times before the while loop terminates.
- Each iteration of loop takes constant time
- Remaining copying operations take linear time

Runtime Analysis: Solve by Tree

On board

Runtime Analysis: Induction

Claim

Mergesort runs in time $O(n \log n)$.

Proof

- Let $T(n)$ be the worst-case runtime of mergesort on arrays of length n.
- There is a constant c such that
  - $T(1) \leq c$
  - $T(n) \leq 2T(n/2) + cn$ for all $n \geq 2$ (why?)
- We can show by strong induction that $T(n) \leq cn(\log(n) + 1)$
- Base case: Trivial
- Inductive step:

$$T(n) = 2T(n/2) + cn \leq 2c\frac{n}{2}(\log(n/2) + 1) + cn = cn(\log n + 1)$$

where the inequality invokes the inductive hypothesis for $n/2$