COP 4531
Complexity & Analysis of Data Structures & Algorithms

Lecture 2
Review of Sorting
The sorting problem

Input: A sequence of \( n \) numbers \(<a_1, a_2, \ldots, a_n>\).
Output: A permutation (reordering) \(<a'_1, a'_2, \ldots, a'_n>\) of the input sequence such that \(a'_1 \leq a'_2 \leq \cdots \leq a'_n\).

• The sequences are typically stored in arrays.
• We refer to the numbers as keys. Along with each key may be additional information that is “sorted along with the key”.
• We will see several ways to solve the sorting problem. Each way will be expressed as an algorithm: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
Insertion sort - in words

It works the way you might sort a hand of playing cards:

- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

- Note that the obvious algorithm (sometimes called the brute force algorithm) is a good place to start when trying to develop a better algorithm.
Pseudo-code for Insertion sort

Insertion-sort (A, n)
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j-1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
Sequence of insertions

(a) 5 2 4 6 1 3
(b) 2 5 4 6 1 3
(c) 2 4 5 6 1 3
(d) 2 4 5 6 1 3
(e) 1 2 4 5 6 3
(f) 1 2 3 4 5 6
Analyzing running time for Insertion sort

Insertion-sort \((A, n)\)

\[
\text{for } j = 2 \text{ to } n \\
\quad \text{key} = A[j] \\
\quad // \text{Insert } A[j] \text{ into the sorted} \\
\quad // \text{sequence } A[1 \ldots j-1] \\
\quad i = j - 1 \\
\quad \text{while } i > 0 \text{ and } A[i] > \text{key} \\
\quad \quad A[i + 1] = A[i] \\
\quad \quad i = i - 1 \\
\quad A[i + 1] = \text{key}
\]

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c_1)</td>
<td>(n)</td>
</tr>
<tr>
<td>(c_2)</td>
<td>(n - 1)</td>
</tr>
<tr>
<td>(c_4)</td>
<td>(n - 1)</td>
</tr>
<tr>
<td>(c_5)</td>
<td>(\sum_{j=2}^{n} t_j)</td>
</tr>
<tr>
<td>(c_6)</td>
<td>(\sum_{j=2}^{n} (t_j - 1))</td>
</tr>
<tr>
<td>(c_7)</td>
<td>(\sum_{j=2}^{n} (t_j - 1))</td>
</tr>
<tr>
<td>(c_8)</td>
<td>(n - 1)</td>
</tr>
</tbody>
</table>

Note that here we assume that each instruction might have a different constant cost. **For** loops and **While** loops also require 1 test in addition to the loop body. Note also that \(t_j\) is the number of times the while loop test is executed for that value of \(j = 2, \ldots, n\). Finally, comments have 0 cost.
The running time equation

The running time $T(n)$ for insertion sort is:

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_8 (n - 1) +$$

$$c_5 \Sigma t_j + c_6 \Sigma (t_j - 1) + c_7 \Sigma (t_j - 1)$$

Best case running time (while loop executed only 1 time each time through loop when array already sorted):

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_8 (n - 1) + c_5 (n - 1)$$

$$= an + b \text{, for constants } a \text{ and } b.$$ 

Thus, in best case, running time is a linear function of $n$. 

How about the worst case?

- Worst case when in reverse sorted order.
- Key is compared with \( j - 1 \) elements so it turns out that \( t_j = j \).
  
  \[
  \sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1
  \]
  
  \[
  \sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1) = \frac{n(n-1)}{2}
  \]

Running time is now:

\[
T(n) = an^2 + bn + c
\]

This is a quadratic function of \( n \)
Usual abstraction:

Consider order of growth

- Drop lower order terms
- Ignore constant coefficient
- Thus, for insertion sort,
  - worst case running time grows as \( n^2 \)
  - The running time is \( \Theta(n^2) \)
Designing Algorithms: Divide and Conquer

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.

2. **Conquer** the subproblems by solving them recursively.
   - **Base case:** If the subproblems are small enough, just solve them by brute force.

3. **Combine** the subproblem solutions to give a solution to the original problem.
Merge sort: uses divide and conquer for sorting

- Worst case running time has a lower growth rate than insertion sort.
- We state each subproblem as sorting a subarray \( A[p .. r] \). Initially, \( p = 1 \) and \( r = n \), but these values change as we recurse through subproblems.
Typical phases for divide and conquer (as applied to merge sort)

- **Divide**: split into two subarrays $A[p .. q]$ and $A[q + 1 .. r]$, where $q$ is the halfway point of $A[p .. r]$.
- **Conquer**: recursively sort the two subarrays $A[p .. q]$ and $A[q + 1 .. r]$.
- **Combine**: merge the two sorted subarrays $A[p .. q]$ and $A[q + 1 .. r]$ to produce a single sorted subarray $A[p .. r]$. To accomplish this step, we define a procedure MERGE ($A$, $p$, $q$, $r$).
- The recursion “bottoms out” when the subarray has just 1 element, so that it’s trivially sorted.
The Merge Procedure

- **Input:** Array $A$ and indices $p$, $q$, $r$ such that:
  
  $p \leq q < r$

  Subarrays $A[p .. q]$ and $A[q + 1 .. r]$ are sorted. Neither array is empty.

- **Output:** The two subarrays are merged into a single sorted subarray in $A[p .. r]$.

- The implementation is $\Theta(n)$ where:

  $n = q - p + 1 + r - q = r - p + 1 = \text{size of the two subarrays}$.

  The merge algorithm is linear in the size of the two subarray.

- By putting a ‘sentinel’ card at the bottom of each subarray, don’t need to keep checking if a subarray is empty (note sentinel value is equal to or larger than any value one would find in the array).
Pseudo Code for 
Merge (A, p, q, r)

Merge (A, p, q, r)
n_1 = q - p + 1
n_2 = r - q
let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
for i = 1 to n_1
    L[i] = A[p + i - 1]
for j = 1 to n_2
    R[j] = A[q + j]
L[n_1 + 1] = \infty
R[n_2 + 1] = \infty
i = 1
j = 1
for k = p to r
    if L[i] \leq R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
Pseudo-code for Merge Sort

Merge-sort(A, p, r)
if p < r
    q = floor(p + r) / 2
    Merge-sort(A, p, q)
    Merge-sort(A, q + 1, r)
    Merge(A, p, q, r)

Initial call: Merge-sort (A, 1, n)
Analyzing Divide and Conquer Algorithms

Use a recurrence equation to describe the running time of a divide and conquer algorithm.

Let $T(n) =$ running time on a problem of size $n$.

- If the problem size is small enough (say, $n \leq c$ for some constant $c$), we have a base case. The brute-force solution takes constant time $\Theta(1)$.

- Otherwise, suppose we divide into $a$ subproblems, each of size $1/b$.
  - Let the divide time for a size $n$ problem be $D(n)$.
  - Solving $a$ subproblems of size $1/b$ takes time $aT(n/b)$.
  - Let the time to combine problems be $C(n)$
General Recurrence

\[ T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c. \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}\]
Analyzing Merge-sort

- For simplicity consider $n$ to be a power of 2. Base case is when $n=1$. Merge is necessary when $n \geq 2$.
- Divide: we simply compute $q$ as the average of $p$ and $r$. Thus $D(n) = \Theta(1)$.
- Conquer: Recursively solve 2 subproblems each of size $n/2$. ($a=2$, $b=2$).
- Combine: MERGE on an $n$-element array takes time $C(n) = \Theta(n)$.

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
2T(n/2) + \Theta(n) & \text{if } n > 1. 
\end{cases}$$

- We can show that the solution to this is:
  \[ T(n) = \Theta(n \lg n) \]
- Compare this to insertion sort which was $\Theta(n^2)$
Another way to understand result

\[ T(n) \]

\[ cn \]

\[ T(n/2) \]

\[ T(n/2) \]

\[ cn/2 \]

\[ T(n/4) \]

\[ T(n/4) \]

\[ cn/2 \]

\[ T(n/4) \]

\[ T(n/4) \]

(a)

(b)

(c)

\[ \text{Total: } cn \lg n + cn \]
Overview of Heapsort

• $O(n \lg n)$ worst case—like merge sort.
• Sorts in place—like insertion sort.
• Combines the best of both algorithms.
• To understand heapsort, we’ll cover heaps and heap operations, and then we’ll take a look at priority queues.
The heap data structure

- A Heap (say A) is a nearly complete binary tree.
  - **Height** of a node = # of edges on a longest simple path from the node down to a leaf.
  - **Height** of the heap = height of root = $\Theta(lg n)$
  - The size of the heap is the number of nodes in the heap (sometimes we use A.heap-size to reference this)
  - A heap could be implemented in many ways such as a binary tree with pointers to nodes or even in an array A of size A.length
The heap and storage in an array

- Root of tree is $A[1]$
- Parent of $A[i] = A\left\lfloor i/2 \right\rfloor$
- Shown is a max-heap with the largest key at the root
The Heap property

• For max-heaps
  - For all nodes i, excluding the root, 
    \( A[\text{Parent}(i)] \geq A[i] \)

• For min-heaps
  - For all nodes i, excluding the root, 
    \( A[\text{Parent}(i)] \leq A[i] \)

• By induction we are guaranteed that the maximum (or minimum for min-heap) element is at the root
“Almost” a heap

• Consider the case where the binary tree is almost a heap except at node $A[i]$ where $A[i]$ is not larger than both of its children.

• We can fix this situation by “moving” the element $A[i]$ down towards the leaves.

• This procedure in the text is called Max-Heapify and essentially heapifies down
Max-Heapify (down)

(a)

(b)

(c)
Max-Heapify (A, i, n)

l = Left (i)

r = Right (i)

if l ≤ n and A[l] > A[i] then largest = l else largest is i

if r ≤ n and A[r] > A[largest] then largest = r

if largest ≠ i

    exchange A[i] with A[largest]

    Max-Heapify (A, largest, n)

Note: the running time on a node of height h is O(h) or O(lg n)
Building a heap from an unordered array: 
\textbf{Build-Max-Heap} (A, n)

\begin{align*}
\text{for } i = \lfloor n/2 \rfloor \text{ downto 1} \\
\text{Max-Heapify} (A, i, n)
\end{align*}

Note: this procedure is clearly $O(n \lg n)$ but in fact a tighter bound of $O(n)$ can be established.
Correctness of Algorithm

• At start of every iteration of the for loop, each node $i + 1, i + 2, \ldots, n$ is a root of a max heap.
• Can be shown by induction.
The heapsort algorithm

• Build a max-heap from the input array
• \(A[1]\) is the max element. Swap with the last element in the array. Reduce the size of the heap by 1.
• Call Max-Heapify on the new root.
• Repeat until only 1 element in heap remains.
Heapsort\((A, n)\)

Build-Max-Heap\((A, n)\)

for \(i = n\) downto \(2\)
  exchange \(A[1]\) with \(A[i]\)
  Max-Heapify\((A, 1, i - 1)\)

This algorithm is \(O(n \lg n)\) since Max-Heapify is \(O(\lg n)\)
Heapsort Example
Overview of Quicksort

• Worst-case running time: $\Theta(n^2)$
• Expected running time: $\Theta(n \lg n)$
• Constants hidden in $\Theta(n \lg n)$ are small
• Sorts in place.
Quicksort(A, p, r)

if p < r
    q = Partition(A, p, r)
    quicksort(A, p, q - 1)
    quicksort(A, q + 1, r)

Partition(A, p, r)

x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
return i + 1
Quicksort example
Basic Analysis:
Worst case / best case

• Worst case partitioning: elements already ordered
  \[ T(n) = T(n-1) + \Theta(n) \]
  running time is \( \Theta(n^2) \)

• Partition is such that at each split, partition is roughly \( \frac{1}{2} \) for each side
  \[ T(n) = 2T(n/2) + \Theta(n) \]
  running time is \( \Theta(n \log n) \)
Average Case is closer to the best case: the intuition

Longest path from root to leaf looking at coefficients is \( r = \frac{9}{10} \):

\[ 1n, r n, r n, r n, \ldots, r n \text{ where } (9/10)^k n = 1 \text{ or } n = (10/9)^k. \]

Thus \( k = \log_{10/9} n \). Note that this depth is still \( c \lg n \).
Lower bound on comparison sorting

• **Comparison sorting**
  - The only operation that may be used to gain order information about a sequence is comparison of pairs of elements.
  - All sorts discussed so far are comparison sorts: insertion sort, mergesort, quicksort, heapsort.
  - We want to show that a lower bound for comparison sort is $n \log n$. We use decision trees to show this.
Example decision tree
Result using decision trees

• Abstract the comparison sort problem by focusing only on comparisons that need to be made
• We note the following:
  - any binary decision tree for comparison sorting must have $n!$ leaves
  - a binary tree of height $h$ has $\leq 2^h$ leaves
  - let $l$ be the number of leaves of a comparison sorting algorithm. Then:
    $$n! \leq l \leq 2^h$$
    $$\log (n!) \leq h$$
  - so the height of any comparison sorting algorithm tree must be at least $\log (n!) = \Theta(n \log n)$
  - Explanation: $\log(n!) = \log(n*(n-1)*(n-2)*...*1) < \log (n^n) = n \log n$
Sorting in linear time

• Counting Sort
  - Key assumption: numbers are integers in \{1, 2, \ldots, k\}
  - $\Theta(n + k)$, which is $\Theta(n)$ if $k = \Theta(n)$
  - This is a stable sort: that is keys with the same value appear in the same order as they did in the input
Counting Sort(A, B, n, k)

let C[0 .. k] be a new array
for i = 0 to k, C[i] = 0
for j = 0 to n  // get counts of each different key values
    C[A[j]] = C[A[j]] + 1
for i = 1 to k  // change counts to key values ≤ i
    C[i] = C[i] + C[i - 1]
for j = n downto 1
    C[A[j]] = C[A[j]] - 1  // decrement location of next similar key
Example of counting sort

A: 2 5 3 0 2 3 0 3
   0 1 2 3 4 5

C: 2 0 2 3 0 1

B: 0 0 0 0 3 3
   0 1 2 3 4 5

C: 1 2 4 6 7 8

C: 1 2 4 5 7 8

A: 2 5 3 0 2 3 0 3

B: 0 0 0 0 3 3
   0 1 2 3 4 5

C: 1 2 4 6 7 8

C: 1 2 4 5 7 8

B: 1 2 3 4 5 6 7 8

C: 2 2 4 6 7 8

B: 1 2 3 4 5 6 7 8

C: 1 2 4 5 7 8

B: 0 0 2 2 3 3 3 5

C: 1 2 4 5 7 8

B: 1 2 3 4 5 6 7 8