8 Sorting in Linear Time

- The sorting algorithms introduced thus far are *comparison sorts*
- Any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort $n$ elements
- Merge sort and heapsort are asymptotically optimal
- We examine counting sort, radix sort, and bucket sort that run in linear time
8.1 Lower bounds for sorting

- We can view comparison sorts abstractly in terms of decision trees – a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm.

- The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf.

- An internal node compares \( a_i \leq a_j \).

- When we come to a leaf, the sorting algorithm has established the ordering \( a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)} \).

- Any correct sorting algorithm must be able to produce each permutation of its input.
  - Each of the \( n! \) permutations must appear as one of the leaves of the decision tree for a comparison sort to be correct.
A lower bound for the worst case

- Length of the longest path from the root to any of the reachable leaves is the worst-case number of comparisons.
- The worst-case number of comparisons for a given algorithm equals the height of its tree.
- A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is a bound on the running time of any comparison sort algorithm.

Theorem 8.1 Any comparison sort algorithm requires \( \Omega(n \lg n) \) comparisons in the worst case.

Proof It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.

Consider a decision tree of height \( h \) with \( \ell \) reachable leaves corresponding to a comparison sort on \( n \) elements.

Each of the \( n! \) permutations of the input appears as some leaf.
Therefore, we have $n! \leq \ell$.

Since a binary tree of height $h$ has no more than $2^h$ leaves, we have

$$n! \leq \ell \leq 2^h$$

which, by taking logarithms, implies $h \geq \log(n!)$ since the $\log$ function is monotonically increasing.

Furthermore, $h = \Omega(n \log n)$, because $\log(n!) = \Theta(n \log n)$

- Hence, heapsort and merge sort are asymptotically optimal comparison sorts

8.2 Counting sort

- Assume that each input element is an integer in the range 0 to $k$, for some integer $k$
- When $k = O(n)$, the sort runs in $\Theta(n)$ time
- Counting sort determines, for an input element $x$, the number of elements $\leq x$
- It uses this information to place element $x$ directly into its position in the output array
- E.g., if 17 elements are less than (or equal to) $x$, then $x$ belongs in output position 18
In addition to the input array $A[1..n]$ we require two other arrays:
- $B[1..n]$ holds the sorted output, and
- $C[0..k]$ provides temporary working storage

**COUNTING-SORT($A, B, k$)
1. let $C[0..k]$ be a new array
2. for $i \leftarrow 0$ to $k$
3. \hspace{1em} $C[i] \leftarrow 0$
4. for $j \leftarrow 1$ to $A.length$
5. \hspace{1em} $C[A[j]] \leftarrow C[A[j]] + 1$
6. \hspace{1em} // $C[i]$ now contains the number of elements equal to $i$
7. for $i \leftarrow 1$ to $k$
8. \hspace{1em} $C[i] \leftarrow C[i] + C[i - 1]$
9. \hspace{1em} // $C[i]$ now contains the number of elements $\leq i$
10. for $j \leftarrow A.length$ downto 1
11. \hspace{1em} $B[C[A[j]]] \leftarrow A[j]$
12. \hspace{1em} $C[A[j]] \leftarrow C[A[j]] - 1$
• How much time does counting sort require?
  – The for loop of lines 2–3 takes time $\Theta(k)$
  – the for loop of lines 4–5 takes time $\Theta(n)$
  – the for loop of lines 7–8 takes time $\Theta(k)$, and
  – the for loop of lines 10–12 takes time $\Theta(n)$
• Thus, the overall time is $\Theta(k + n)$
• In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$

• Counting sort beats the lower bound of $\Omega(n \log n)$ because it is not a comparison sort
• In fact, no comparisons between input elements occur anywhere in the code
• Instead, counting sort uses the actual values of the elements to index into an array
• An important property of counting sort is that it is **stable**
  – numbers with the same value appear in the output array in the same order as they do in the input array
8.3 Radix sort

- Radix sort is used by the card-sorting machines you now find only in computer museums
- Radix sort solves the problem of card sorting counter-intuitively by sorting on the least significant digit first
- In order for radix sort to work correctly, the digit sorts must be stable

Radix-Sort\((A, d)\)
1. for \(i \leftarrow 1\) to \(d\)
2. use a stable sort to sort array \(A\) on digit \(i\)
Lemma 8.3  Given \( n \) \( d \)-digit numbers in which each digit can take on up to \( k \) possible values, RADIX-SORT sorts the numbers in \( \Theta(d(k + n)) \) time if the stable sort takes \( \Theta(k + n) \) time.

**Proof**  The correctness of radix sort follows by induction on the columns. When each digit is in the range 0 to \( k - 1 \) (so that it can take on \( k \) possible values), and \( k \) is not too large, counting sort is the obvious choice. Each pass over \( n \) \( d \)-digit numbers then takes time \( \Theta(k + n) \). There are \( d \) passes, and so the total time for radix sort follows.

---

8.4 Bucket sort

- Assume that the input is drawn from uniform distribution and has an average-case running time of \( O(n) \).
- Counting and bucket sort are fast because they assumes something about the input:
  - Counting sort: the input contains integers in a small range,
  - Bucket sort: the input is drawn from a random process that distributes elements uniformly and independently over the interval \([0,1)\).
Bucket sort divides the interval \([0, 1]\) into \(n\) equal-sized subintervals, or buckets, and then distributes the \(n\) input numbers into the buckets. Since inputs are uniformly and independently distributed over \([0, 1]\), we do not expect many numbers to fall into each bucket. We sort numbers in each bucket and go through the buckets in order, listing the elements in each. The input is an \(n\)-element array \(A\) and each element \(A[i]\) in the array satisfies \(0 \leq A[i] < 1\). The code requires an auxiliary array \(B[0..n-1]\) of linked lists (buckets).

**BUCKET-SORT** \((A)\)

1. let \(B[0..n-1]\) be a new array
2. \(n \leftarrow A.length\)
3. for \(i \leftarrow 0\) to \(n - 1\)
   4. make \(B[i]\) an empty list
5. for \(i \leftarrow 1\) to \(n\)
   6. insert \(A[i]\) into list \(B[nA[i]]\)
7. for \(i \leftarrow 0\) to \(n - 1\)
   8. sort list \(B[i]\) with insertion sort
9. concatenate the lists \(B[0], B[1], \ldots, B[n - 1]\) together in order
Consider two elements $A[i]$ and $A[j]$

- Assume wlog that $A[i] \leq A[j]$
- Since $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, either element $A[i]$ goes into the same bucket as $A[j]$ or it goes into a bucket with a lower index
- If $A[i]$ and $A[j]$ go into the same bucket
  - the for loop of lines 7–8 puts them into the proper order
- If $A[i]$ and $A[j]$ go into different buckets
  - line 9 puts them into the proper order
- Therefore, bucket sort works correctly
• Observe that all lines except line 8 take $O(n)$ time in the worst case
• We need to analyze the total time taken by the $n$ calls to insertion sort in line 8
• Let $n_i$ be the random variable denoting the number of elements placed in bucket $B[i]$
• Since insertion sort runs in quadratic time, the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

We now compute the expected value of the running time, where we take the expectation over the input distribution
• Taking expectations of both sides and using linearity of expectation, we have

$$\mathbb{E}[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} \mathbb{E}[O(n_i^2)]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(\mathbb{E}[n_i^2])$$
• Claim: $E[n_i^2] = 2 - 1/n$ for $i = 0,1,\ldots,n-1$

• It is no surprise that each bucket $i$ has the same value of $E[n_i^2]$, since each value in $A$ is equally likely to fall in any bucket

• To prove the claim, define indicator random variables $X_{ij} = I\{A[j] \text{ falls in bucket } i\}$ for $i = 0,1,\ldots,n-1$ and $j = 1,2,\ldots,n$

• Thus,

$$n_i = \sum_{j=1}^{n} X_{ij}$$

$$E[n_i^2] = E \left[ \left( \sum_{j=1}^{n} X_{ij} \right)^2 \right] = E \left[ \sum_{j=1}^{n} \sum_{k=1}^{n} X_{ij}X_{ik} \right]$$

$$= E \left[ \sum_{j=1}^{n} X_{ij}^2 + \sum_{1\leq j\leq n} \sum_{1\leq k\leq n, k\neq j} X_{ij}X_{ik} \right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{1\leq k\leq n, k\neq j} E[X_{ij}X_{ik}]$$
• Indicator random variable $X_{ij}$ is 1 with probability $1/n$ and 0 otherwise, therefore
  \[ E[X_{ij}^2] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n} \]

• When $k \neq j$, the variables $X_{ij}$ and $X_{ik}$ are independent, and hence
  \[ E[X_{ij}X_{ik}] = E[X_{ij}] E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2} \]

• Substituting these two expected values, we obtain
  \[ E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} \frac{1}{n^2} \]
  \[ = n \cdot \frac{1}{n} + n(n - 1) \cdot \frac{1}{n^2} \]
  \[ = 1 + \frac{n - 1}{n} = 2 - \frac{1}{n} \]

• which proves the claim
• Using this expected value we conclude that average-case running time for bucket sort is 
\[ \Theta(n) + n \cdot O(2 - 1/n) = \Theta(n) \]

• Even if the input is not drawn uniformly, bucket sort may still run in linear time
• As long as the input has the property
  – the sum of the squares of the bucket sizes is linear in the total number of elements,
• bucket sort will run in linear time

9 Medians and Order Statistics

• The \( i \)th order statistic of a set of \( n \) elements is the \( i \)th smallest element
  – E.g., the minimum of a set of elements is the first order statistic \( (i = 1) \), and the maximum is the \( n \)th order statistic \( (i = n) \)
• A median is the “halfway point” of the set
• When \( n \) is odd, the median is unique, occurring at \( i = (n + 1)/2 \)
When \( n \) is even, there are two medians, occurring at \( i = n/2 \) and \( i = n/2 + 1 \)

Thus, regardless of the parity of \( n \), medians occur at

\[-\left\lfloor \frac{i}{2} \right\rfloor (\text{the lower median}) \text{ and} \]

\[-\left\lceil \frac{i}{2} \right\rceil (\text{the upper median}) \]

For simplicity, we use “the median” to refer to the lower median

The problem of selecting the \( i \)th order statistic from a set of \( n \) distinct numbers

We assume that the set contains distinct numbers

– virtually everything extends to the situation in which a set contains repeated values

We formally specify the problem as follows:

– **Input:** A set \( A \) of \( n \) (distinct) numbers and an integer \( i \), with \( 1 \leq i \leq n \)

– **Output:** The element \( x \in A \) that is larger than exactly \( i - 1 \) other elements of \( A \)
• We can solve the problem in $O(n \log n)$ time by heapsort or merge sort and then simply index the $i$th element in the output array.
• There are faster algorithms.
• First, we examine the problem of selecting the minimum and maximum of a set of elements.
• Then we analyze a practical randomized algorithm that achieves an $O(n)$ expected running time, assuming distinct elements.

9.1 Minimum and maximum

• How many comparisons are necessary to determine the minimum of a set of $n$ elements?
• We can easily obtain an upper bound of $n - 1$ comparisons:
  – examine each element of the set in turn and keep track of the smallest element seen so far.
• In the following procedure, we assume that the set resides in array $A$, where $A.length = n$. 
\[
\text{MINIMUM}(A)
\]
1. \(\text{min} \leftarrow A[1]\)
2. \(\text{for } i \leftarrow 2 \text{ to } A.length\)
3. \(\text{if } \text{min} > A[i]\)
4. \(\text{min} \leftarrow A[i]\)
5. \(\text{return } \text{min}\)

- We can, of course, find the max with \(n - 1\) comparisons as well

- This is the best we can do, since we can obtain a lower bound of \(n - 1\) comparisons
  - Think of any algorithm that determines the minimum as a tournament among the elements
  - Each comparison is a match in the tournament in which the smaller of the two elements wins
  - Observing that every element except the winner must lose at least one match, we conclude that \(n - 1\) comparisons are necessary to determine the minimum

- Hence, the algorithm MINIMUM is optimal w.r.t. the number of comparisons performed
**Simultaneous minimum and maximum**

- Sometimes, we must find both the minimum and the maximum of a set of $n$ elements.
- For example, a graphics program may need to scale a set of $(x, y)$ data to fit onto a rectangular display screen or other graphical output device.
- To do so, the program must first determine the minimum and maximum value of each coordinate.

- $\Theta(n)$ comparisons is asymptotically optimal:
  - Simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2$ comparisons.
  - In fact, we can find both the minimum and the maximum using at most $3 \lfloor n/2 \rfloor$ comparisons by maintaining both the minimum and maximum elements seen thus far.
  - Rather than processing each element of the input by comparing it against the current minimum and maximum, we process elements in pairs.
• Compare pairs of input elements first with each other, and then we compare the smaller with the current min and the larger to the current max, at a cost of 3 comparisons for every 2 elements
• If $n$ is odd, we set both the min and max to the value of the first element, and then we process the rest of the elements in pairs
• If $n$ is even, we perform 1 comparison on the first 2 elements to determine the initial values of the min and max, and then process the rest of the elements in pairs as in the case for odd $n$

• If $n$ is odd, then we perform $3\lceil n/2 \rceil$ comparisons
• If $n$ is even, we perform 1 initial comparison followed by $3(n - 2)/2$ comparisons, for a total of $3n/2 - 2$
• Thus, in either case, the total number of comparisons is at most $3\lceil n/2 \rceil$
9.2 Selection in expected linear time

- The selection problem appears more difficult than finding a minimum, but the asymptotic running time for both is the same: $\Theta(n)$
- A divide-and-conquer algorithm \textsc{Randomized-Select} is modeled after the quicksort algorithm
- Unlike quicksort, \textsc{Randomized-Select} works on only one side of the partition
- Whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected running time of \textsc{Randomized-Select} is $\Theta(n)$, assuming that the elements are distinct

\textsc{Randomized-Select} uses the procedure \textsc{Randomized-Partition} of \textsc{Randomized-Quicksort}

\textsc{Randomized-Partition}(A, p, r)
1. $i \leftarrow \text{Random}(p, r)$
2. exchange $A[r]$ with $A[i]$
3. return \textsc{Partition}(A, p, r)
Partitioning the array

**PARTITION**\((A, p, r)\)

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

---

**Diagram:**

(a)\[\begin{array}{cccccc}
  i & p & j & r \\
  \hline
  2 & 8 & 7 & 1 & 3 & 5 & 6 & 4
\end{array}\]

(b)\[\begin{array}{cccccc}
  \p & \i & j & r \\
  \hline
  2 & 8 & 7 & 1 & 3 & 5 & 6 & 4
\end{array}\]

(c)\[\begin{array}{cccccc}
  \p & \i & j & r \\
  \hline
  2 & 8 & 7 & 1 & 3 & 5 & 6 & 4
\end{array}\]

(d)\[\begin{array}{cccccc}
  \p & i & j & r \\
  \hline
  2 & 8 & 7 & 1 & 3 & 5 & 6 & 4
\end{array}\]

(e)\[\begin{array}{cccccc}
  \p & i & j & r \\
  \hline
  2 & 1 & 7 & 8 & 3 & 5 & 6 & 4
\end{array}\]

(f)\[\begin{array}{cccccc}
  p & i & j & r \\
  \hline
  2 & 1 & 3 & 8 & 7 & 5 & 6 & 4
\end{array}\]

(g)\[\begin{array}{cccccc}
  p & i & j & r \\
  \hline
  2 & 1 & 3 & 8 & 7 & 5 & 6 & 4
\end{array}\]

(h)\[\begin{array}{cccccc}
  p & i & j & r \\
  \hline
  2 & 1 & 3 & 8 & 7 & 5 & 6 & 4
\end{array}\]

(i)\[\begin{array}{cccccc}
  p & i & j & r \\
  \hline
  2 & 1 & 3 & 4 & 7 & 5 & 6 & 8
\end{array}\]
• **PARTITION** always selects an element \( x = A[r] \) as a *pivot* element around which to partition the subarray \( A[p..r] \).

• As the procedure runs, it partitions the array into four (possibly empty) regions.

• At the start of each iteration of the for loop in lines 3–6, the regions satisfy properties, shown above.

- At the beginning of each iteration of the loop of lines 3–6, for any array index \( k \),
  1. If \( p \leq k \leq i \), then \( A[k] \leq x \)
  2. If \( i + 1 \leq k \leq j - 1 \), then \( A[k] > x \)
  3. If \( k = r \), then \( A[k] = x \)

• Indices between \( j \) and \( r - 1 \) are not covered by any case, and the values in these entries have no particular relationship to the pivot \( x \).

• The running time of **PARTITION** on the subarray \( A[p..r] \) is \( \Theta(n) \), \( n = p - r + 1 \).
• Return the \(i\)th smallest element of \(A[p..r]\)

**RANDOMIZED-SELECT**(\(A,p,r,i\))

1. if \(p = r\)
   2. return \(A[p]\)
3. \(q \leftarrow \text{RANDOMIZED-PARTITION}(A,p,r)\)
4. \(k \leftarrow q - p + 1\)
5. if \(i = k\)  // the pivot value is the answer
   6. return \(A[q]\)
7. elseif \(i < k\)
   8. return \(\text{RANDOMIZED-SELECT}(A,p,q-1,i)\)
9. else return \(\text{RANDOMIZED-SELECT}(A,q+1,r,i-k)\)

(1) checks for the base case of the recursion

Otherwise, \(\text{RANDOMIZED-PARTITION}\) partitions \(A[p..r]\) into two (possibly empty) subarrays \(A[p..q-1]\) and \(A[q+1..r]\) s.t. each element in the former is \(\leq A[q]\) < each element of the latter

(4) computes the \(k\) of elements in \(A[p..q]\)

(5) checks if \(A[q]\) is \(i\)th smallest element

Otherwise, determine in which of the two subarrays the \(i\)th smallest element lies

If \(i < k\), then the desired element lies on the low side of the partition, and (8) recursively selects it
• If \( i > k \), then the desired element lies on the high side of the partition

• Since we already know \( k \) values that are smaller than the \( i \)th smallest element of \( A[p..r] \) the desired element is the \( (i - k) \)th smallest element of \( A[q + 1..r] \), which (9) finds recursively

Worst-case running time for RANDOMIZED-SELECT is \( \Theta(n^2) \), even to find the minimum

• The algorithm has a linear expected running time, though

• Because it is randomized, no particular input elicits the worst-case behavior

• Let the running time of RANDOMIZED-SELECT on an input array \( A[p..r] \) of \( n \) elements be a random variable \( T(n) \)

• We obtain an upper bound on \( E[T(n)] \) as follows
• The procedure RANDOMIZED-PARTITION is equally likely to return any element as the pivot
• Therefore, for each \( k \) such that \( 1 \leq k \leq n \), the subarray \( A[p \ldots q] \) has \( k \) elements (all less than or equal to the pivot) with probability \( 1/n \)
• For \( k = 1, 2, \ldots, n \), define indicator random variables \( X_k \) where

\[
X_k = \mathbb{I}\{\text{the subarray } A[p \ldots q] \text{ has exactly } k \text{ elements}\}
\]

• Assuming that the elements are distinct, we have \( \mathbb{E}[X_k] = 1/n \)
• When we choose \( A[q] \) as the pivot element, we do not know, \textit{a priori},
  1) if we will terminate immediately with the correct answer,
  2) recurse on the subarray \( A[p \ldots q - 1] \), or
  3) recurse on the subarray \( A[q + 1 \ldots r] \)
• This decision depends on where the \( i \)th smallest element falls relative to \( A[q] \)
• Assuming $T(n)$ to be monotonically increasing, we can upper-bound the time needed for a recursive call by that on largest possible input.

• To obtain an upper bound, we assume that the $i$th element is always on the side of the partition with the greater number of elements.

• For a given call of RANDOMIZED-SELECT, the indicator random variable $X_k$ has value 1 for exactly one value of $k$, and it is 0 for all other $k$.

• When $X_k = 1$, the two subarrays on which we might recurse have sizes $k - 1$ and $n - k$.

We have the recurrence

$$T(n) \leq \sum_{i=1}^{n} X_k \cdot (T(\max(k - 1, n - k)) + O(n))$$

$$= \sum_{i=1}^{n} X_k \cdot T(\max(k - 1, n - k)) + O(n)$$

• Taking expected values, we have

$$E[T(n)] \leq E\left[\sum_{i=1}^{n} X_k \cdot T(\max(k - 1, n - k)) + O(n)\right]$$

$$= \sum_{i=1}^{n} E[X_k \cdot T(\max(k - 1, n - k))] + O(n)$$
The first Eq. on this slide follows by independence of random variables $X_k$ and $\max(k-1, n-k)$.

Consider the expression

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lfloor n/2 \rfloor \\ n-k & \text{if } k \leq \lfloor n/2 \rfloor \end{cases}$$

If $n$ is even, each term from $T(\lfloor n/2 \rfloor)$ up to $T(n-1)$ appears exactly twice in the summation, and if $n$ is odd, all these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once.

Thus, we have

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n)$$

We can show that $E[T(n)] = O(n)$ by substitution.

In summary, we can find any order statistic, and in particular the median, in expected linear time, assuming that the elements are distinct.