Introduction to Computer Programming
LECTURE 5

Sorting Lower Bounds
• Decision trees

Linear-Time Sorting
• Counting sort
• Radix sort

Order Statistics
• Randomized divide and conquer
• Analysis of expected time
• Worst-case linear-time order statistics
• Analysis
How fast can we sort?

All the sorting algorithms we have seen so far are *comparison sorts*: only use comparisons to determine the relative order of elements.

- *E.g.*, insertion sort, merge sort, quicksort, heapsort.

The best worst-case running time that we’ve seen for comparison sorting is $O(n \lg n)$.

*Is $O(n \lg n)$ the best we can do?*

*Decision trees* can help us answer this question.
Decision-tree example

Sort $\langle a_1, a_2, \ldots, a_n \rangle$

Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$. 
- The left subtree shows subsequent comparisons if $a_i \leq a_j$. 
- The right subtree shows subsequent comparisons if $a_i \geq a_j$. 
Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$. 
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$. 

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$: 

```
1:2
2:3       1:3
123       9 ≥ 6
132       312
213       231
321
```
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.
Sort $\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$:

Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ has been established.
A decision tree can model the execution of any comparison sort:

- One tree for each input size $n$.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.
Lower bound for decision-tree sorting

**Theorem.** Any decision tree that can sort \( n \) elements must have height \( \Omega(n \lg n) \).

**Proof.** The tree must contain \( \geq n! \) leaves, since there are \( n! \) possible permutations. A height-\( h \) binary tree has \( \leq 2^h \) leaves. Thus, \( n! \leq 2^h \).

\[
\therefore h \geq \lg(n!)
\geq \lg ((n/e)^n)
= n \lg n - n \lg e
= \Omega(n \lg n).
\]
Lower bound for comparison sorting

**Corollary.** Heapsort and merge sort are asymptotically optimal comparison sorting algorithms.
Sorting in linear time

Counting sort: No comparisons between elements.

• **Input**: \( A[1 \ldots n] \), where \( A[j] \in \{1, 2, \ldots, k\} \).
• **Output**: \( B[1 \ldots n] \), sorted.
• **Auxiliary storage**: \( C[1 \ldots k] \).
Counting sort

for $i \leftarrow 1$ to $k$
    do $C[i] \leftarrow 0$
for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$  \hspace{1em} \triangleright C[i] = |\{\text{key} = i\}|
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$  \hspace{1em} \triangleright C[i] = |\{\text{key} \leq i\}|
for $j \leftarrow n$ downto $1$
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$
Counting-sort example

\begin{align*}
A & : \quad 4 \quad 1 \quad 3 \quad 4 \quad 3 \\
B & : & \\
C & : & 1 \quad 2 \quad 3 \quad 4
\end{align*}
Loop 1

for $i \leftarrow 1$ to $k$
  do $C[i] \leftarrow 0$
Loop 2

\[
\begin{array}{c|ccccc}
A & 1 & 2 & 3 & 4 & 5 \\
\hline
4 & 1 & 3 & 4 & 3 & 1 \\
\end{array}
\]

\[
\begin{array}{c|ccccc}
B & 1 & 2 & 3 & 4 & 5 \\
\hline
\end{array}
\]

\[
\begin{array}{c|ccccc}
C & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright C[i] = |\{\text{key} = i\}|
\]
Loop 2

for $j \leftarrow 1$ to $n$
\hspace{1cm} do $C[A[j]] \leftarrow C[A[j]] + 1$
\hspace{1cm} $\triangleright C[i] = |\{\text{key} = i\}|$
Loop 2

\[
\begin{align*}
A: & \quad \begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3 \\
\end{array} \\
B: & \quad \begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array} \\
C: & \quad \begin{array}{cccc}
1 & 0 & 1 & 1 \\
\end{array}
\end{align*}
\]

\textbf{for } j \leftarrow 1 \textbf{ to } n
\textbf{ do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}|
Loop 2

\[\text{for } j \leftarrow 1 \text{ to } n\]
\[\text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}|\]
Loop 2

\[\begin{array}{ccccc}
A: & 1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3
\end{array}\]

\[\begin{array}{ccccc}
B: & & & & & \\
& & & & &
\end{array}\]

\[\begin{array}{ccccc}
C: & 1 & 2 & 3 & 4 \\
1 & 0 & 2 & 2
\end{array}\]

\textbf{for} \ j \leftarrow 1 \ \textbf{to} \ n \\
\textbf{do}\ C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}|
Loop 3

\[
\begin{align*}
A: & \quad \begin{bmatrix}
4 & 1 & 3 & 4 & 3 \\
\end{bmatrix} \\
B: & \quad \begin{bmatrix}
\end{bmatrix} \\
C: & \quad \begin{bmatrix}
1 & 0 & 2 & 2 \\
\end{bmatrix} \\
C': & \quad \begin{bmatrix}
1 & 1 & 2 & 2 \\
\end{bmatrix}
\end{align*}
\]

\[
\text{for } i \leftarrow 2 \text{ to } k \\
\quad \text{do } C[i] \leftarrow C[i] + C[i-1] \quad \triangleright \quad C[i] = |\{\text{key} \leq i\}|
\]
Loop 3

\[\]  

\[\begin{array}{c|c|c|c|c|c} & 1 & 2 & 3 & 4 & 5 \\ \hline A: & 4 & 1 & 3 & 4 & 3 \\ \end{array}\]  

\[\begin{array}{c|c|c|c|c} & 1 & 2 & 3 & 4 \\ \hline B: & & & & \\ \end{array}\]  

\[\begin{array}{c|c|c|c|c} & 1 & 0 & 2 & 2 \\ \hline C: & & & & \\ \end{array}\]  

\[\begin{array}{c|c|c|c|c} & 1 & 1 & 3 & 2 \\ \hline C': & & & & \\ \end{array}\]  

\textbf{for} \(i \leftarrow 2\ \textbf{to} \ k\)  
\textbf{do} \(C[i] \leftarrow C[i] + C[i-1]\)  
\(\triangleright \ C[i] = |\{\text{key} \leq i\}|\)
Loop 3

\[
\begin{align*}
A: & \quad 1 & 2 & 3 & 4 & 5 \\
& \quad \begin{array}{|c|c|c|c|c|}
& 4 & 1 & 3 & 4 & 3 \\
\end{array} \\
B: & \quad \begin{array}{|c|c|c|c|c|}
& & & & & \\
\end{array} \\
C: & \quad 1 & 0 & 2 & 2 \\
C': & \quad 1 & 1 & 3 & 5 \\
\end{align*}
\]

\textbf{for} \ i \leftarrow 2 \ \textbf{to} \ k \quad \textbf{do} \ C[i] \leftarrow C[i] + C[i-1] \quad \triangleright \ C[i] = |\{\text{key} \leq i\}|
Loop 4

\begin{align*}
\text{for } j &\leftarrow n \text{ downto } 1 \\
\text{do } B[C[A[j]]] &\leftarrow A[j] \\
C[A[j]] &\leftarrow C[A[j]] - 1
\end{align*}
Loop 4

\[
\begin{array}{c}
A: \\
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 3 & 4 & 3 \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{c}
B: \\
\begin{array}{ccccc}
1 & 2 & 3 & 4 \\
3 & 4 \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{c}
C: \\
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 1 & 2 & 5 \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{c}
C': \\
\begin{array}{ccccc}
1 & 2 & 3 & 4 \\
1 & 1 & 2 & 4 \\
\end{array} \\
\end{array}
\]

for \( j \leftarrow n \) downto 1

\[
\begin{array}{c}
C[A[j]] \leftarrow C[A[j]] - 1 \\
\end{array}
\]
Loop 4

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
A: & 4 & 1 & 3 & 4 & 3 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 \\
B: & 3 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 1 & 2 & 4 \\
C: & 1 & 1 & 1 & 4 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 1 & 2 & 4 \\
C': & 1 & 1 & 1 & 4 \\
\end{array}
\]

\[\text{for } j \leftarrow n \text{ downto } 1 \]
\[\text{do } B[C[A[j]]] \leftarrow A[j] \]
\[C[A[j]] \leftarrow C[A[j]] - 1 \]
Loop 4

for $j \leftarrow n \text{ downto } 1$

\[ B[C[A[j]]] \leftarrow A[j] \]
\[ C[A[j]] \leftarrow C[A[j]] - 1 \]
Loop 4

\[
\begin{align*}
A &: \begin{bmatrix} 4 & 1 & 3 & 4 & 3 \end{bmatrix} \\
B &: \begin{bmatrix} 1 & 3 & 3 & 4 & 4 \end{bmatrix} \\
C &: \begin{bmatrix} 0 & 1 & 1 & 4 \end{bmatrix} \\
C' &: \begin{bmatrix} 0 & 1 & 1 & 3 \end{bmatrix}
\end{align*}
\]

\[
\text{for } j \leftarrow n \text{ downto } 1 \\
\text{do } B[C[A[j]]] \leftarrow A[j] \\
C[A[j]] \leftarrow C[A[j]] - 1
\]
Analysis

\( \Theta(k) \)
\[
\begin{align*}
&\text{for } i \leftarrow 1 \text{ to } k \\
&\quad \text{do } C[i] \leftarrow 0
\end{align*}
\]

\( \Theta(n) \)
\[
\begin{align*}
&\text{for } j \leftarrow 1 \text{ to } n \\
&\quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1
\end{align*}
\]

\( \Theta(k) \)
\[
\begin{align*}
&\text{for } i \leftarrow 2 \text{ to } k \\
&\quad \text{do } C[i] \leftarrow C[i] + C[i-1]
\end{align*}
\]

\( \Theta(n) \)
\[
\begin{align*}
&\text{for } j \leftarrow n \text{ downto } 1 \\
&\quad \text{do } B[C[A[j]]] \leftarrow A[j] \quad C[A[j]] \leftarrow C[A[j]] - 1
\end{align*}
\]

\( \Theta(n + k) \)
Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where’s the fallacy?

**Answer:**

- *Comparison sorting* takes $\Omega(n \lg n)$ time.
- Counting sort is not a *comparison sort*.
- In fact, not a single comparison between elements occurs!
Stable sorting

Counting sort is a \textit{stable} sort: it preserves the input order among equal elements.

\begin{itemize}
  \item \textbf{A}: 4 1 3 4 3
  \item \textbf{B}: 1 3 3 4 4
\end{itemize}

\textbf{Exercise}: What other sorts have this property?
Radix sort

• **Origin**: Herman Hollerith’s card-sorting machine for the 1890 U.S. Census. (See Appendix 1.)

• Digit-by-digit sort.

• Hollerith’s original (bad) idea: sort on most-significant digit first.

• Good idea: Sort on *least-significant digit first* with auxiliary *stable* sort.
# Operation of radix sort

<table>
<thead>
<tr>
<th>3 2 9</th>
<th>7 2 0</th>
<th>7 2 0</th>
<th>3 2 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 7</td>
<td>3 5 5</td>
<td>3 2 9</td>
<td>3 5 5</td>
</tr>
<tr>
<td>6 5 7</td>
<td>4 3 6</td>
<td>4 3 6</td>
<td>4 3 6</td>
</tr>
<tr>
<td>8 3 9</td>
<td>4 5 7</td>
<td>8 3 9</td>
<td>4 5 7</td>
</tr>
<tr>
<td>4 3 6</td>
<td>6 5 7</td>
<td>3 5 5</td>
<td>6 5 7</td>
</tr>
<tr>
<td>7 2 0</td>
<td>3 2 9</td>
<td>4 5 7</td>
<td>7 2 0</td>
</tr>
<tr>
<td>3 5 5</td>
<td>8 3 9</td>
<td>6 5 7</td>
<td>8 3 9</td>
</tr>
</tbody>
</table>
Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit $t$
Correctness of radix sort

**Induction on digit position**

- Assume that the numbers are sorted by their low-order $(t-1)$ digits.

- Sort on digit $t$
  - Two numbers that differ in digit $t$ are correctly sorted.
Correctness of radix sort

Induction on digit position

• Assume that the numbers are sorted by their low-order \( t - 1 \) digits.

• Sort on digit \( t \)
  ▪ Two numbers that differ in digit \( t \) are correctly sorted.
  ▪ Two numbers equal in digit \( t \) are put in the same order as the input \( \Rightarrow \) correct order.
Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort \( n \) computer words of \( b \) bits each.
- Each word can be viewed as having \( b/r \) base-\( 2^r \) digits.

Example: 32-bit word

\[
\begin{array}{cccc}
8 & 8 & 8 & 8 \\
\end{array}
\]

\( r = 8 \Rightarrow b/r = 4 \) passes of counting sort on base-\( 2^8 \) digits; or \( r = 16 \Rightarrow b/r = 2 \) passes of counting sort on base-\( 2^{16} \) digits.

How many passes should we make?
Recall: Counting sort takes $\Theta(n + k)$ time to sort $n$ numbers in the range from 0 to $k - 1$. If each $b$-bit word is broken into $r$-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are $b/r$ passes, we have

$$T(n, b) = \Theta\left(\frac{b}{r} \left(n + 2^r\right)\right).$$

Choose $r$ to minimize $T(n, b)$:

- Increasing $r$ means fewer passes, but as $r \gg \lg n$, the time grows exponentially.
Choosing $r$

$$T(n, b) = \Theta\left(\frac{b}{r} (n + 2^r)\right)$$

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don’t want $2^r \gg n$, and there’s no harm asymptotically in choosing $r$ as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(bn/\lg n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.
Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):
• At most 3 passes when sorting \( \geq 2000 \) numbers.
• Merge sort and quicksort do at least \( \lceil \lg 2000 \rceil = 11 \) passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.
Order statistics

Select the $i$th smallest of $n$ elements (the element with rank $i$).

- $i = 1$: minimum;
- $i = n$: maximum;
- $i = \left\lfloor \frac{(n+1)/2}{2} \right\rfloor$ or $\left\lceil \frac{(n+1)/2}{2} \right\rceil$: median.

**Naive algorithm**: Sort and index $i$th element. Worst-case running time $= \Theta(n \ lg n) + \Theta(1) = \Theta(n \ lg n)$, using merge sort or heapsort (not quicksort).
Randomized divide-and-conquer algorithm

\textbf{RAND-SELECT}(A, p, q, i) \quad \triangleright \quad \text{i}^{\text{th}} \text{ smallest of } A[p \ldots q]

\textbf{if} \quad p = q \quad \textbf{then return} \quad A[p]

r \leftarrow \text{RAND-PARTITION}(A, p, q)

k \leftarrow r - p + 1 \quad \triangleright \quad k = \text{rank}(A[r])

\textbf{if} \quad i = k \quad \textbf{then return} \quad A[r]

\textbf{if} \quad i < k

\quad \textbf{then return} \quad \text{RAND-SELECT}(A, p, r - 1, i)

\textbf{else return} \quad \text{RAND-SELECT}(A, r + 1, q, i - k)
Example

Select the $i = 7$th smallest:

\[
\begin{array}{cccccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
\end{array}
\]

pivot

Partition:

\[
\begin{array}{cccccccccc}
2 & 5 & 3 & 6 & 8 & 13 & 10 & 11 \\
\end{array}
\]

Select the $7 - 4 = 3$rd smallest recursively.
Intuition for analysis

(All our analyses today assume that all elements are distinct.)

Lucky:
\[ T(n) = T(9n/10) + \Theta(n) \]
\[ = \Theta(n) \]

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

Worse than sorting!

\[ n^{\log_{10} 9} = n^0 = 1 \]

Case 3

arithmetic series
Analysis of expected time

The analysis follows that of randomized quicksort, but it’s a little different.

Let $T(n) =$ the random variable for the running time of **RAND-SELECT** on an input of size $n$, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the *indicator random variable*

\[
X_k = \begin{cases} 
1 & \text{if } \text{PARTITION} \text{ generates a } k : n-k-1 \text{ split,} \\
0 & \text{otherwise.}
\end{cases}
\]
Analysis (continued)

To obtain an upper bound, assume that the $i$th element always falls in the larger side of the partition:

$$T(n) = \begin{cases} 
  T(\max\{0, n-1\}) + \Theta(n) & \text{if } 0 : n-1 \text{ split}, \\
  T(\max\{1, n-2\}) + \Theta(n) & \text{if } 1 : n-2 \text{ split}, \\
  \vdots \\
  T(\max\{n-1, 0\}) + \Theta(n) & \text{if } n-1 : 0 \text{ split}, 
\end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left( T(\max\{k, n-k-1\}) + \Theta(n) \right).$$
Calculating expectation

\[ E[T(n)] = E\left[ \sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right] \]

Take expectations of both sides.
Calculating expectation

\[ E[T(n)] = E\left[ \sum_{k=0}^{n-1} X_k \left( T(\max\{k, n-k-1\}) + \Theta(n) \right) \right] \]

\[ = \sum_{k=0}^{n-1} E\left[ X_k \left( T(\max\{k, n-k-1\}) + \Theta(n) \right) \right] \]

Linearity of expectation.
Calculating expectation

\[ E[T(n)] = E\left[ \sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right] \]

\[ = \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)] \]

Independence of \( X_k \) from other random choices.
Calculating expectation

\[
E[T(n)] = E\left[ \sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right]
\]

\[
= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)]
\]

\[
= \frac{1}{n} \sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)
\]

Linearity of expectation; \( E[X_k] = 1/n \).
Calculating expectation

\[ E[T(n)] = E \left[ \sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right] \]

\[ = \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)] \]

\[ = \frac{1}{n} \sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \]

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

Upper terms appear twice.
Hairy recurrence

(But not quite as hairy as the quicksort one.)

\[ E[T(n)] = \frac{2}{n} \sum_{k=[n/2]}^{n-1} E[T(k)] + \Theta(n) \]

Prove: \( E[T(n)] \leq cn \) for constant \( c > 0 \).

- The constant \( c \) can be chosen large enough so that \( E[T(n)] \leq cn \) for the base cases.

Use fact: \( \sum_{k=[n/2]}^{n-1} k \leq \frac{3}{8} n^2 \) (exercise).
Substitution method

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

Substitute inductive hypothesis.
Substitution method

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

\leq \frac{2c}{n} \left( \frac{3}{8} n^2 \right) + \Theta(n)

Use fact.
Substitution method

\[ E[T(n)] \leq 2 \sum_{n/2}^{n-1} c_k + \Theta(n) \]

\[ \leq \frac{2c}{n} \left( \frac{3}{8} n^2 \right) + \Theta(n) \]

\[ = cn - \left( \frac{cn}{4} - \Theta(n) \right) \]

Express as \textit{desired} – \textit{residual}. 
Substitution method

\[ E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} c k + \Theta(n) \]
\[ \leq \frac{2c}{n} \left( \frac{3}{8} n^2 \right) + \Theta(n) \]
\[ = cn - \left( \frac{cn}{4} - \Theta(n) \right) \]
\[ \leq cn, \]

if \( c \) is chosen large enough so that \( cn/4 \) dominates the \( \Theta(n) \).
Summary of randomized order-statistic selection

- Works fast: linear expected time.
- Excellent algorithm in practice.
- But, the worst case is very bad: $\Theta(n^2)$.

**Q.** Is there an algorithm that runs in linear time in the worst case?

**A.** Yes, due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973].

**Idea:** Generate a good pivot recursively.
Worst-case linear-time order statistics

**SELECT**(*i*, *n*)

1. Divide the *n* elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively **SELECT** the median *x* of the \( \lfloor n/5 \rfloor \) group medians to be the pivot.
3. Partition around the pivot *x*. Let *k* = \( \text{rank}(x) \).
4. **if** *i* = *k** then return *x***
   **elseif** *i* < *k***
   **then** recursively **SELECT** the *i*th smallest element in the lower part
   **else** recursively **SELECT** the \((i-k)\)th smallest element in the upper part

Same as **RAND-SELECT**
Choosing the pivot
Choosing the pivot

1. Divide the $n$ elements into groups of 5.
Choosing the pivot

1. Divide the $n$ elements into groups of 5. Find the median of each 5-element group by rote.
Choosing the pivot

1. Divide the $n$ elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median $x$ of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
At least half the group medians are \( \leq x \), which is at least \( \left\lfloor \frac{n}{5} \right\rfloor / 2 = \left\lfloor \frac{n}{10} \right\rfloor \) group medians.
At least half the group medians are $\leq x$, which is at least $\lceil \lfloor n/5 \rfloor /2 \rceil = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3 \lfloor n/10 \rfloor$ elements are $\leq x$. 

(Assume all elements are distinct.)
At least half the group medians are $\leq x$, which is at least $\left\lfloor \frac{n}{5} \right\rfloor /2 = \left\lfloor \frac{n}{10} \right\rfloor$ group medians.

- Therefore, at least $3 \left\lfloor \frac{n}{10} \right\rfloor$ elements are $\leq x$.
- Similarly, at least $3 \left\lfloor \frac{n}{10} \right\rfloor$ elements are $\geq x$. 

(Assume all elements are distinct.)
Minor simplification

- For $n \geq 50$, we have $3 \lfloor n/10 \rfloor \geq n/4$.
- Therefore, for $n \geq 50$ the recursive call to SELECT in Step 4 is executed recursively on $\leq 3n/4$ elements.
- Thus, the recurrence for running time can assume that Step 4 takes time $T(3n/4)$ in the worst case.
- For $n < 50$, we know that the worst-case time is $T(n) = \Theta(1)$.
Developing the recurrence

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th>$\text{SELECT}(i, n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>1. Divide the $n$ elements into groups of 5. Find the median of each 5-element group by rote.</td>
</tr>
<tr>
<td>$T(n/5)$</td>
<td>2. Recursively $\text{SELECT}$ the median $x$ of the $\lfloor n/5 \rfloor$ group medians to be the pivot.</td>
</tr>
<tr>
<td>$\Theta(n)$</td>
<td>3. Partition around the pivot $x$. Let $k = \text{rank}(x)$.</td>
</tr>
</tbody>
</table>
| $T(3n/4)$ | 4. if $i = k$ then return $x$  
elseif $i < k$  
then recursively $\text{SELECT}$ the $i$th smallest element in the lower part  
else recursively $\text{SELECT}$ the $(i-k)$th smallest element in the upper part |

$T(n) = \Theta(n)$
Solving the recurrence

\[ T(n) = T\left(\frac{1}{5} n\right) + T\left(\frac{3}{4} n\right) + \Theta(n) \]

**Substitution:**

\[
T(n) \leq \frac{1}{5} cn + \frac{3}{4} cn + \Theta(n)
\]

\[
= \frac{19}{20} cn + \Theta(n)
\]

\[
= cn \left( 1 - \frac{1}{20} \right) cn - \Theta(n) \]

\[
\leq cn ,
\]

if \( c \) is chosen large enough to handle both the \( \Theta(n) \) and the initial conditions.
Conclusions

• Since the work at each level of recursion is a constant fraction (19/20) smaller, the work per level is a geometric series dominated by the linear work at the root.

• In practice, this algorithm runs slowly, because the constant in front of $n$ is large.

• The randomized algorithm is far more practical.

Exercise: Why not divide into groups of 3?