CS61B Lectures #27

Today:

• Merge sorts
• Quicksort

Readings: Today: DS(IJ), Chapter 8; Next topic: Chapter 9.
Merge Sorting

Idea: Divide data in 2 equal parts; recursively sort halves; merge results.

- Already seen analysis: $\Theta(N \lg N)$.
- Good for external sorting:
  - First break data into small enough chunks to fit in memory and sort.
  - Then repeatedly merge into bigger and bigger sequences.
- Can merge $K$ sequences of arbitrary size on secondary storage using $\Theta(K)$ storage:

```java
Data[] V = new Data[K];
For all i, set V[i] to the first data item of sequence i;
while there is data left to sort:
    Find k so that V[k] has data and is smallest;
    Add V[k] to output sequence;
    If there is more data in sequence k, read it into V[k],
    otherwise, clear V[k];
```

Last modified: Tue Oct 26 15:35:37 2021
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain lists, initially empty.
- Bucket $#k$ is either empty or contains $2^k$ sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length $2^k$ into bucket $k$. Whenever that gives a list of size $2^{k+1}$, merge it into bucket $k+1$ and clear bucket $k$.
- When all inputs are processed, merge all the buckets into the final list.

$L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)$

\[
\begin{array}{c}
0: & 0 \\
1: & 0 \\
2: & 0 \\
3: & 0 \\
\end{array}
\begin{array}{c}
\text{Merge} \\
\end{array}
\]

(9)
Illustration of Internal Merge Sort

For internal sorting, can use a *binomial comb* to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain lists, initially empty.
- Bucket $#k$ is either empty or contains $2^k$ sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length $2^k$ into bucket $k$. Whenever that gives a list of size $2^{k+1}$, merge it into bucket $k + 1$ and clear bucket $k$.
- When all inputs are processed, merge all the buckets into the final list.

$L$: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

\[0: \begin{array}{c} 1 \end{array} \rightarrow (9)\]
\[1: \begin{array}{c} 0 \end{array}\]
\[2: \begin{array}{c} 0 \end{array}\]
\[3: \begin{array}{c} 0 \end{array}\]
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain lists, initially empty.
- Bucket $\#k$ is either empty or contains $2^k$ sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length $2^k$ into bucket $k$. Whenever that gives a list of size $2^{k+1}$, merge it into bucket $k + 1$ and clear bucket $k$.
- When all inputs are processed, merge all the buckets into the final list.

$L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)$

```
0: [1]  (9)  Merge  (15)
1: [0]
2: [0]
3: [0]
```
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain lists, initially empty.
- Bucket #\(k\) is either empty or contains $2^k$ sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length $2^k$ into bucket $k$. Whenever that gives a list of size $2^{k+1}$, merge it into bucket $k+1$ and clear bucket $k$.
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

\[
\begin{array}{c}
0: \\ 1: \\ 2: \\ 3: \\
\end{array}
\quad \text{Merge} \quad (9, 15)
\]
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with $\lg N + 1$ buckets that can contain lists, initially empty.
- Bucket #k is either empty or contains $2^k$ sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length $2^k$ into bucket $k$. Whenever that gives a list of size $2^{k+1}$, merge it into bucket $k + 1$ and clear bucket $k$.
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with \( \lg N + 1 \) buckets that can contain lists, initially empty.
- Bucket \( \#k \) is either empty or contains \( 2^k \) sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length \( 2^k \) into bucket \( k \). Whenever that gives a list of size \( 2^{k+1} \), merge it into bucket \( k + 1 \) and clear bucket \( k \).
- When all inputs are processed, merge all the buckets into the final list.

\[
L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)
\]

```
0: 0
1: 1 - (9, 15)
2: 0
3: 0
```

\[ \text{Merge} \]
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with \( \lg N + 1 \) buckets that can contain lists, initially empty.
- Bucket \( \#k \) is either empty or contains \( 2^k \) sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length \( 2^k \) into bucket \( k \). Whenever that gives a list of size \( 2^{k+1} \), merge it into bucket \( k + 1 \) and clear bucket \( k \).
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with \( \lg N + 1 \) buckets that can contain lists, initially empty.
- Bucket \( \#k \) is either empty or contains \( 2^k \) sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length \( 2^k \) into bucket \( k \). Whenever that gives a list of size \( 2^{k+1} \), merge it into bucket \( k + 1 \) and clear bucket \( k \).
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

```
0: [1]  Merge (5) (3)
1: [1]  (9, 15)
2: [0]  
3: [0]  
```
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with \( \lg N + 1 \) buckets that can contain lists, initially empty.
- Bucket \( \#k \) is either empty or contains \( 2^k \) sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length \( 2^k \) into bucket \( k \). Whenever that gives a list of size \( 2^{k+1} \), merge it into bucket \( k + 1 \) and clear bucket \( k \).
- When all inputs are processed, merge all the buckets into the final list.

\[ L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8) \]

```
0: [ ]
1: [1]
2: [0]
3: [0]
```

\[ (9, 15) \rightarrow (3, 5) \]
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with \( \lg N + 1 \) buckets that can contain lists, initially empty.
- Bucket \( \#k \) is either empty or contains \( 2^k \) sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length \( 2^k \) into bucket \( k \). Whenever that gives a list of size \( 2^{k+1} \), merge it into bucket \( k + 1 \) and clear bucket \( k \).
- When all inputs are processed, merge all the buckets into the final list.

\[
L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)
\]

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

\[
\text{Merge} \quad (3, 5, 9, 15)
\]
Illustration of Internal Merge Sort

For internal sorting, can use a binomial comb to orchestrate an iterative merge sort.

- Start with \( \lg N + 1 \) buckets that can contain lists, initially empty.
- Bucket \( \#k \) is either empty or contains \( 2^k \) sorted items at any time.
- For each item in the input list, turn it into a 1-element list, and merge it into bucket 0 (or simply put it in bucket 0 if that is empty).
- You will only merge lists of length \( 2^k \) into bucket \( k \). Whenever that gives a list of size \( 2^{k+1} \), merge it into bucket \( k + 1 \) and clear bucket \( k \).
- When all inputs are processed, merge all the buckets into the final list.

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

0: \[
\begin{array}{c}
0 \\
1 \\
2 \\
3
\end{array}
\]

1: \[
\begin{array}{c}
0 \\
0 \\
1
\end{array}
\]

2: \[
\begin{array}{c}
0 \\
1 \\
0
\end{array}
\]

3: \[
\begin{array}{c}
0 \\
1 \\
0
\end{array}
\]

Merge (0) (3, 5, 9, 15)
Illustration of Internal Merge Sort (II)

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)

0: 0
1: 0
2: 0
3: 0

0 elements processed

0: 1 → 9
1: 0
2: 0
3: 0

1 element processed

0: 0
1: 1 → (9, 15)
2: 0
3: 0

2 elements processed

0: 1 → (0, 6)
1: 1 → (3, 5, 9, 15)
2: 0
3: 0

3 elements processed

0: 1 → (8)
1: 1 → (2, 20)
2: 0
3: 1 → (-1, 0, 3, 5, 6, 9, 10, 15)

11 elements processed

Final Step: Merge all the lists into (-1, 0, 2, 3, 5, 6, 8, 9, 10, 15, 20)
Quicksort: Speed through Probability

Idea:

- **Partition** data into pieces: everything > a *pivot* value at the high end of the sequence to be sorted, and everything ≤ on the low end.

- Repeat recursively on the high and low pieces.

- For speed, stop when pieces are “small enough” and do insertion sort on the whole thing.

- Reason: insertion sort has low constant factors. By design, no item will move out of its piece [why?], so when pieces are small, #inversions is, too.

- Have to choose pivot well. E.g.: *median* of first, last and middle items of sequence.
Example of Quicksort

- In this example, we continue until pieces are size $\leq 4$.
- Pivots for next step are starred. Arrange to move pivot to dividing line each time.
- Last step is insertion sort.

<table>
<thead>
<tr>
<th>16</th>
<th>10</th>
<th>13</th>
<th>18</th>
<th>-4</th>
<th>-7</th>
<th>12</th>
<th>-5</th>
<th>19</th>
<th>15</th>
<th>0</th>
<th>22</th>
<th>29</th>
<th>34</th>
<th>-1*</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>-5</td>
<td>-7</td>
<td>-1</td>
<td>18</td>
<td>13</td>
<td>12</td>
<td>10</td>
<td>19</td>
<td>15</td>
<td>0</td>
<td>22</td>
<td>29</td>
<td>34</td>
<td>16*</td>
</tr>
<tr>
<td>-4</td>
<td>-5</td>
<td>-7</td>
<td>-1</td>
<td>15</td>
<td>13</td>
<td>12*</td>
<td>10</td>
<td>0</td>
<td>16</td>
<td>19*</td>
<td>22</td>
<td>29</td>
<td>34</td>
<td>18</td>
</tr>
<tr>
<td>-4</td>
<td>-5</td>
<td>-7</td>
<td>-1</td>
<td>10</td>
<td>0</td>
<td>12</td>
<td>15</td>
<td>13</td>
<td>16</td>
<td>18</td>
<td>19</td>
<td>29</td>
<td>34</td>
<td>22</td>
</tr>
</tbody>
</table>

- Now everything is “close to” right, so just do insertion sort:

| -7 | -5 | -4 | -1 | 0 | 10 | 12 | 13 | 15 | 16 | 18 | 19 | 22 | 29 | 34  |
Performance of Quicksort

• Probabalistic time:
  - If choice of pivots good, divide data in two each time: $\Theta(N \lg N)$ with a good constant factor relative to merge or heap sort.
  - If choice of pivots bad, most items on one side each time: $\Theta(N^2)$.
  - $\Omega(N \lg N)$ in best case, so insertion sort better for nearly ordered input sets.

• Interesting point: randomly shuffling the data before sorting makes $\Omega(N^2)$ time very unlikely!
Quick Selection

The Selection Problem: for given $k$, find $k^{th}$ smallest element in data.

- Obvious method: sort, select element #$_k$, time $\Theta(N \lg N)$.
- If $k \leq$ some constant, can easily do in $\Theta(N)$ time:
  - Go through array, keep smallest $k$ items.
- Get probably $\Theta(N)$ time for all $k$ by adapting quicksort:
  - Partition around some pivot, $p$, as in quicksort, arrange that pivot ends up at dividing line.
  - Suppose that in the result, pivot is at index $m$, all elements $\leq$ pivot have indices $\leq m$.
  - If $m = k$, you’re done: $p$ is answer.
  - If $m > k$, recursively select $k^{th}$ from left half of sequence.
  - If $m < k$, recursively select $(k - m - 1)^{th}$ from right half of sequence.
Selection Example

Problem: Find just item #10 in the sorted version of array:

Initial contents:

```
  51 | 60 | 21 | -4 | 37 | 4 | 49 | 10 | 40* | 59 | 0 | 13 | 2 | 39 | 11 | 46 | 31
```

Looking for #10 to left of pivot 40:

```
   13 | 31 | 21 | -4 | 37 | 4* | 11 | 10 | 39 | 2 | 0 | 40 | 59 | 51 | 49 | 46 | 60
```

Looking for #6 to right of pivot 4:

```
  -4 | 0 | 2 | 4 | 37 | 13 | 11 | 10 | 39 | 21 | 31* | 40 | 59 | 51 | 49 | 46 | 60
```

Looking for #1 to right of pivot 31:

```
  -4 | 0 | 2 | 4 | 21 | 13 | 11 | 10 | 31 | 39 | 37 | 40 | 59 | 51 | 49 | 46 | 60
```

Just two elements; just sort and return #1:

```
  -4 | 0 | 2 | 4 | 21 | 13 | 11 | 10 | 31 | 37 | 39 | 40 | 59 | 51 | 49 | 46 | 60
```

Result: 39
Selection Performance

- For this algorithm, if \( m \) roughly in middle each time, cost is

\[
C(N) = \begin{cases} 
1, & \text{if } N = 1, \\
N + C(N/2), & \text{otherwise.}
\end{cases}
\]

\[
= N + N/2 + \ldots + 1
\]

\[
= 2N - 1 \in \Theta(N)
\]

- But in worst case, get \( \Theta(N^2) \), as for quicksort.

- By another, non-obvious algorithm, can get \( \Theta(N) \) worst-case time for all \( k \) (take CS170).