Running times continued

- some running times are more difficult to analyze

**Merging two sorted lists**

**Input:** Two arrays $A = \{a_1, a_2, \ldots, a_m\}$, $B = \{b_1, b_2, \ldots, b_n\}$, in increasing order

**Output:** Array $C$ containing $A \cup B$, in increasing order

\[
A = \{1, 5, 6, 7, 10, 11\} \\
B = \{2, 4, 8, 9, 15\}
\]
Merging two sorted lists

Input: Two arrays $A = \{a_1, a_2, \ldots, a_m\}$, $B = \{b_1, b_2, \ldots, b_n\}$, in increasing order

Output: Array $C$ containing $A \cup B$, in increasing order

MERGE($A,B$)
1. $i=1$; $j=1$; $k=1$;
2. $a_{m+1}=\infty$; $b_{n+1}=\infty$;
3. while ($k \leq m+n$) do
4. if ($a_i < b_j$) then
5. $c_k=a_i$; $i++$;
6. else
7. $c_k=b_j$; $j++$;
8. $k++$
9. RETURN $C=\{c_1, c_2, \ldots, c_{m+n}\}$

Running time $O(m+n)$
Running times continued

**Sorting**

Input: An array \( X = \{x_1, x_2, \ldots, x_n\} \)

Output: \( X \) sorted in increasing order
Running times continued

**Sorting**

Input: An array $X = \{x_1, x_2, \ldots, x_n\}$

Output: $X$ sorted in increasing order

*MergeSort* – a *divide-and-conquer* algorithm

MERGESORT($X, n$)
1. if ($n == 1$) RETURN $X$
2. $middle = n/2$ (round down)
3. $A = \{x_1, x_2, \ldots, x_{\text{middle}}\}$
4. $B = \{x_{\text{middle}+1}, x_{\text{middle}+2}, \ldots, x_n\}$
5. $As = \text{MERGESORT}(A, middle)$
6. $Bs = \text{MERGESORT}(B, n-mid\text{dle})$
7. RETURN $\text{MERGE}(As, Bs)$
Running times continued

## Sorting

**Input:** An array $X = \{x_1, x_2, \ldots, x_n\}$

**Output:** $X$ sorted in increasing order

### MergeSort

`MERGESORT(X,n)`

1. if ($n == 1$) RETURN $X$
2. $\text{middle} = \lfloor n/2 \rfloor$ (round down)
3. $A = \{x_1, x_2, \ldots, x_{\text{middle}}\}$
4. $B = \{x_{\text{middle}+1}, x_{\text{middle}+2}, \ldots, x_n\}$
5. $A_s = \text{MERGESORT}(A,\text{middle})$
6. $B_s = \text{MERGESORT}(B,n-\text{middle})$
7. RETURN $\text{MERGE}(A_s,B_s)$

Running time?

$T(n)$

where $n$ is the input

$T(n) = 2T(\frac{n}{2}) + O(n)$

$\forall n \geq 2$

$T(1) = O(1)$
A recurrence

Running time of MergeSort: \( T(n) \)

How to bound \( T(n) \)?

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad \text{if } n > 1 \]
\[ T(1) = O(1) \]

\( \rightarrow \) “unrolling the recurrence”

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + cn \quad \text{if } n > 1 \]
\[ T(n) \leq c \quad \text{if } n \leq 1 \]

\[ T(n) \leq 2T\left(\frac{n}{2}\right) + cn \leq 2\left(2T\left(\frac{n}{4}\right) + c\cdot\frac{n}{2}\right) + cn = 4T\left(\frac{n}{4}\right) + 2cn \leq 4\left(2T\left(\frac{n}{8}\right) + c\cdot\frac{n}{4}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn \leq \ldots = 2^k T\left(\frac{n}{2^k}\right) + kcn \]

Stop when \( \frac{n}{2^k} \leq 1 \), i.e. for \( k = \log n \)

\[ T(n) \leq n \cdot c + \log n \cdot c \cdot n = O(n \log n) \]

Summing this: depth \( (c+d) \cdot n \)

Depth \( \leq \log n \) to get to single elements

\[ \Rightarrow O(n \log n) \text{ steps} \]
A recurrence

Running time of MergeSort: $T(n)$

How to bound $T(n)$?

$\Rightarrow$ “substitution / induction”

$T(n) \leq 2T(\frac{n}{2}) + cn$ if $n > 1$

$T(n) \leq c$ if $n \leq 1$

Guess that $T(n) \leq d \cdot n \cdot \log n$ for $n \geq 2$ (we still need to figure out what $d$ is)

Verify by induction:

**Base Case:** $T(2) \leq 2T(1) + 2c \leq 2c + 2c = 4c$

Want $4c \leq d \cdot 2 \cdot \log 2 = 2d$ (we need $d \geq 2c$)

**Inductive Case:** Want to verify that $T(n) = d \cdot n \cdot \log n$ for $n \geq 2$ assuming $T(n/2) \leq d \cdot \frac{n}{2} \cdot \log \frac{n}{2}$ (inductive hypothesis).

$T(n) \leq 2T(n/2) + cn$

By hypothesis:

$\leq 2 \cdot d \cdot \frac{n}{2} \cdot \log \frac{n}{2} + cn = d \cdot n \cdot (\log n - 1) + cn$

Want to show $\leq d \cdot n \cdot \log n$.

holds $\forall c (d \geq 2c)$
More on sorting

Other $O(n \log n)$ sorts?
Can do better than $O(n \log n)$?
HeapSort
- underlying datastructure: heap

Def: A heap is a complete binary tree, with nodes storing keys, and the property that for every parent and child:

\[ \text{key(parent)} \leq \text{key(child)}. \]
More on sorting

HeapSort
- underlying datastructure: heap

Use: priority queue - a datastructure that supports:
- extract-min
- add key
- change key value

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linked List</th>
<th>Sorted Linked List</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract-min</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>add key</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>change key value</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
More on sorting

Heap
- stored in an array - how to compute:

- how to add a key?

Parent(i) = ⌊(i-1)/2⌋
Left child(i) = 2i + 1
Right child(i) = 2i + 2

e.g. adding key 4.5

heap: depth = ⌈log(n+1)⌉
More on sorting

Heap
- stored in an array - how to compute:

- how to add a key?

**HEAPIFY-UP**(*H*, *i*)
1. while (*i* > 0) and (*H*[*i*] < *H*[Parent(*i*)]) do
2. swap entries *H*[*i*] and *H*[Parent(*i*)]
3. *i* = Parent(*i*)

**ADD**(*H*, key)
1. *H*[H.length] = key
2. H.length++
3. HEAPIFY-UP(*H*, H.length)
More on sorting

Heap
- stored in an array - how to compute:
  - Parent(i) = (i-1)/2
  - LeftChild(i) = 2i+1
  - RightChild(i) = 2i+2

- what if we change the value of a key (at position i)?
  - if key decreased, then: \textsc{Heapify-Up}
  - otherwise?

  "bubble" down
  \rightarrow swap with
  the smaller child
  (if the child
  is < new value)
More on sorting

Heap
- stored in an array - how to compute:

Parent(i) = \((i-1)/2\)
LeftChild(i) = \(2i+1\)
RightChild(i) = \(2i+2\)

- what if we change the value of a key (at position \(i\))?

HEAPIFY-DOWN(H,i)
1. \(n = H.length\)
2. while (LeftChild(i)<n and \(H[i] > H[LeftChild(i)]\)) or (RightChild(i)<n and \(H[i] > H[RightChild(i)]\)) do
3. if (\(H[LeftChild(i)] < H[RightChild(i)]\)) then
4. \(j = LeftChild(i)\)
5. else
6. \(j = RightChild(i)\)
7. swap entries \(H[i]\) and \(H[j]\)
8. \(i = j\)
More on sorting

Heap
- running times:

Use: **priority queue** - a datastructure that supports:
- extract-min
- add key
- change key value
More on sorting

HeapSort

HEAPSORT(A)
1. H = BUILD_HEAP(A)
2. n = A.length
3. for i=0 to n-1 do
4. A[i] = EXTRACT_MIN(H)

BUILD_HEAP(A)
1. initially H = ∅
2. n = A.length
3. for i=0 to n-1 do
4. ADD(H,A[i])

EXTRACT_MIN(H)
1. min = H[0]
2. H.length--
3. H[0] = H[H.length]
4. HEAPIFY_DOWN(H,0)
5. RETURN min

Note (more efficient BUILD_HEAP):
A different implementation of BUILD_HEAP runs in time O(n).
More on sorting

HeapSort

HEAPSORT(A)
1. H = BUILD_HEAP(A)
2. n = A.length
3. for i=0 to n-1 do
4. A[i] = EXTRACT_MIN(H)

BUILD_HEAP(A)
1. initially H = ∅
2. n = A.length
3. for i=0 to n-1 do
4. ADD(H,A[i])

EXTRACT_MIN(H)
1. min = H[0]
2. H.length--
3. H[0] = H[H.length]
4. HEAPIFY_DOWN(H,0)
5. RETURN min

Running time:
Related data structures

Balanced binary search tree

Inserting 8

Different types of balanced binary search trees:
1) Lookup of an element: $O(\log n)$
2) Add/Delete: $O(\log n)$

Rotate the tree left of 5 to balance it out:
A lower-bound on sorting: $\Omega(n \log n)$

Every comparison-based sort needs at least $\Omega(n \log n)$ comparisons, thus it’s running time is $\Omega(n \log n)$.

1) depth is equal to
   the # of comparisons
   needed to sort the input
   (in the worst case)

2) if two inputs are
different orders of
the same numbers,
they end up in different
leaves
(bec. we make the
same operations if
they follow the same
path down the decision tree)

3) hence: #leaves is $\geq n!$
   for a binary tree with $k$ leaves, its depth
   $\geq \log k$

4) last step:
   # comparisons $\geq \log(n!)
   = \Theta(n \log n)$
Sorting faster than $O(n \log n)$?

We know:
Every comparison-based sort needs at least $\Omega(n \log n)$ comparisons.

Can we possibly sort faster than $O(n \log n)$?
**RadixSort** - a non-comparison based sort.

**Idea:** First sort the input by the last digit.

```
275
727
327
273
377
737
```

Next we get:

```
727
327
737
273
275
377
```

Sort by the last digit:

```
273
275
727
327
377
737
```

Sort by the 2nd to last digit:

```
273 → 275
727 → 327 → 377 → 737
```

Each iteration takes $O(n)$ steps (assuming that addition to a linked list is constant).

**Running time:**

$O(dn)$ where $d$ is the digit in the largest number.
RadixSort - a non-comparison based sort.

RADIXSORT(A)
1. d = length of the longest element in A
2. for j=1 to d do
3.   COUNTSORT(A,j) // a stable sort to sort A
      // by the j-th last digit

COUNTSORT (A,j)
1. let B[0..9] be an array of (empty) linked-lists
2. n = A.length
3. for i=0 to n-1 do
4.   let x be the j-th last digit of A[i]
5.   add A[i] at the end of the linked-list B[x]
6. copy B[1] followed by B[2], then B[3], etc. to A

Running time? O(dn)