Running times continued

- some running times are more difficult to analyze

**Merging two sorted lists**

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

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

Merging two sorted lists

Input: Two arrays $A = \{a_1, a_2, ..., a_m\}, B = \{b_1, b_2, ..., b_n\}$, 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. \hspace{1cm} $c_k=a_i; i++$;
6. else
7. \hspace{1cm} $c_k=b_j; j++$;
8. \hspace{1cm} $k++$;
9. RETURN $C=\{c_1, c_2, ..., c_{m+n}\}$

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

\[ X = 2 \quad 5 \quad 1 \quad 8 \quad 3 \quad 6 \quad 4 \quad 7 \]

- Recursively sort
- Merge
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_{middle}\}
4. B = \{x_{middle+1}, x_{middle+2}, \ldots, x_n\}
5. As = MERGESORT(A,middle)
6. Bs = MERGESORT(B,n-middle)
7. RETURN 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**

\[
\text{MERGESORT}(X,n) \quad T(n) - \text{the running time for input of size } n
\]

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. \(A_s = \text{MERGESORT}(A, \text{middle})\) \(T(\frac{n}{2})\)
6. \(B_s = \text{MERGESORT}(B, n-\text{middle})\) \(T(\frac{n}{2})\)
7. RETURN \(\text{MERGE}(A_s, B_s)\)

Running time?

\[
T(n) \leq c \cdot n + 2T(\frac{n}{2})
\]

for steps 1, 2, 3, \ldots, \(n\)

We will see it is \(O(n \log n)\).
A recurrence

Running time of MergeSort: $T(n) \leq 2T\left(\frac{n}{2}\right) + c \cdot n$ $\forall n \geq 2$ \hspace{1cm} (3)

$T(1) = c$ $\hspace{1cm} n=1$ \hspace{1cm} (4)

How to bound $T(n)$?

$\rightarrow$ “unrolling the recurrence”

$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n \\ = 2 \cdot (2T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}) + c \cdot n \\ = 4T\left(\frac{n}{4}\right) + 2c \cdot n$ \hspace{1cm} (5)

$= 4 \cdot (2T\left(\frac{n}{8}\right) + c \cdot \frac{n}{4}) + 2c \cdot n = 8T\left(\frac{n}{8}\right) + 3c \cdot n$ \hspace{1cm} (6)

$= 8 \cdot (2T\left(\frac{n}{16}\right) + c \cdot \frac{n}{8}) + 3c \cdot n = 16T\left(\frac{n}{16}\right) + 4c \cdot n$ \hspace{1cm} (7)

$= 2^k \cdot T\left(\frac{n}{2^k}\right) + k \cdot c \cdot n$ \hspace{1cm} (8)

$= 2^k \cdot c + k \cdot c \cdot n = n \cdot c + c \cdot n \cdot \log_2 n$ \hspace{1cm} \text{(assume we applied (8) \ $k$ times)}

we stop applying (8) when $\frac{n}{2^k} = 1 \hspace{1cm} (k = \log_2 n)$

\hspace{1cm} therefore, $T(n) = c \cdot \log_2 n + c \cdot n = O(n \log n)$
A recurrence

Running time of MergeSort: \( T(n) \leq 2T\left(\frac{n}{2}\right) + c \cdot n \)

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

\( T(1) \leq c \)

\( \forall n \geq 2 \)

-> “substitution / induction”

**Claim:** There exists a constant \( d > 0 \) s.t. \( T(n) \leq d \cdot n \cdot \log n \) \( \forall n \geq \frac{1}{2}, \ n \) is a power of 2

**Proof by induction on \( n \):**

**BASE CASE:** \( n = \frac{1}{2} \): we need to show \( T\left(\frac{1}{2}\right) \leq d \cdot \frac{1}{2} \cdot \log \frac{1}{2} \)

- We know: \( T(2) \leq 2T\left(\frac{2}{2}\right) + c \cdot 2 \leq 2c + 2c = 4c \)
- RHS: \( d \cdot 2 \cdot \log 2 = 2d \) so LHS \( \leq \) RHS if \( 4c \leq 2d \)

**IND. CASE:** \( n \), a power of 2, \( \geq 2 \)

We need to show: \( T(n) \leq d \cdot n \cdot \log n \)

**INDUCTIVE HYPOTHESIS:** \( T\left(\frac{n}{2}\right) \leq d \cdot \frac{n}{2} \cdot \log \frac{n}{2} \)

\( T(n) \leq 2T\left(\frac{n}{2}\right) + c \cdot n \leq 2 \cdot d \cdot \frac{n}{2} \cdot \log \frac{n}{2} + c \cdot n = d \cdot n \cdot (\log n - 1) + c \cdot n = d \cdot n \log n - d \cdot n + c \cdot n \leq d \cdot n \log n \)

\( \Box \)
More on sorting

Other $O(n \log n)$ sorts?

Can do better than $O(n \log n)$?
More on sorting

**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 \( O(\log n) \)
- add key \( O(\log n) \)
- change key value \( O(\log n) \)

Side comment:
if implementing the priority queue using linked list (or an array):
- extract min \( O(n) \)
- add key \( O(1) \)
- change value \( O(1) \)
More on sorting

Heap
- stored in an array - how to compute:

- how to add a key?

Parent[i] = \lfloor i/2 \rfloor
Left child[i] = 2i
Right child[i] = 2i + 1

\[ \begin{array}{c}
1 & 2 & 3 & 4 & 5 & 6 \\
\downarrow & & & & & \\
2 & 3 & 5 & 10 & 9 & 8
\end{array} \]
More on sorting

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

- how to add a key?

HEAPIFY-UP(H,i)
1. while (i > 1) 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.size++
2. H[H.size] = key
3. HEAPIFY-UP(H,H.size)
More on sorting

Heap
- stored in an array - how to compute:
  \[ \text{Parent}(i) = \frac{i}{2} \]
  \[ \text{LeftChild}(i) = 2i \]
  \[ \text{RightChild}(i) = 2i + 1 \]

- what if we change the value of a key (at position \( i \))?
  - if key decreased, then: \text{HEAPIFY-UP}
  - otherwise? \text{HEAPIFY-DOWN}
More on sorting

**Heap**
- stored in an array - how to compute:
  \[
  \text{Parent}(i) = \lfloor i/2 \rfloor \\
  \text{LeftChild}(i) = 2i \\
  \text{RightChild}(i) = 2i + 1
  \]

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

**HEAPIFY-DOWN** (H, i)
1. \( n = H\text{.size} \)
2. while \( (\text{LeftChild}(i) \leq n \text{ and } H[i] > H[\text{LeftChild}(i)]) \)
   or \( (\text{RightChild}(i) \leq n \text{ and } H[i] > H[\text{RightChild}(i)]) \) do
3.   if \( (H[\text{LeftChild}(i)] < H[\text{RightChild}(i)]) \) then
4.     \( j = \text{LeftChild}(i) \)
5.   else
6.     \( j = \text{RightChild}(i) \)
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=1 to n do
4. A[i] = EXTRACT_MIN(H)

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

EXTRACT_MIN(H)
1. min = H[1]
3. H.size--
4. HEAPIFY_DOWN(H, 1)
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

For any node, its value
\[ \geq \text{the value of any node in the left subtree} \]
\[ \leq \text{right subtree} \]

In the worst case, run. time of search is \( O(n) \)

If balanced (the depth is \( O(\log n) \))
then run. time \( O(\log n) \)

Moreover:
- \( O(\log n) \) search, add, remove
- Needs rebalancing using tree rotations
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 $\Omega(n \log n)$. 
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 $O(n \log n)$?
Sorting faster than $O(n \log n)$?

**RadixSort** - a non-comparison based sort.

Idea: First sort the input by the last digit.
Sorting faster than $O(n \log n)$?

**RadixSort** - a non-comparison based sort.

**RADIIXSORT** (A)

1. $d = \text{length of the longest element in } A$
2. for $j=1$ to $d$ do
3. \hspace{1cm} **COUNTSORT** (A,j) // a stable sort to sort A
   \hspace{1cm} // by the $j$-th last digit

**COUNTSORT** (A,j)

1. let $B[0..9]$ be an array of (empty) linked-lists
2. $n = A.\text{length}$
3. for $i=0$ to $n-1$ do
4. \hspace{1cm} let $x$ be the $j$-th last digit of $A[i]$
5. \hspace{1cm} add $A[i]$ at the end of the linked-list $B[x]$

**Running time?**
Sorting faster than $O(n \log n)$?

**RadixSort** - a non-comparison based sort.

Idea: First sort the input by the last digit.
**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. \( \text{COUNTSORT}(A, j) \) // 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=1 to n 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?