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

\[
\begin{array}{cccccc}
& & & 3 & 7 & 8 & 15 \\
1 & & & & & \\
& & & 4 & 5 & 9 & 17 & 22 \\
2 & & & & & \\
& & & 1 & 2 & & \\
\end{array}
\]
Running times continued

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 = 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

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

Running time of MergeSort:  \( T(n) = \begin{cases} 
0(1) & \text{if } n=1 \\
T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) & \text{if } n > 1 
\end{cases} \)

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

-> “unrolling the recurrence”

rounding the recurrence (not use \( O(\cdot) \))

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

for \( n=1 \)

for \( n > 1 \) \((*)\)

(technically speaking, this should be \( n_0 \))

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + c \cdot n \leq 2 \left[ 2 \cdot T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2} \right] + c \cdot n = 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot c \cdot n \leq 4 \cdot \left[ 2 \cdot T\left(\frac{n}{8}\right) + c \cdot \frac{n}{4} \right] + 2c \cdot n
\]

= \( 8 \cdot T\left(\frac{n}{8}\right) + 3c \cdot n \leq 8 \left[ 2 \cdot T\left(\frac{n}{16}\right) + c \cdot \frac{n}{8} \right] + 3c \cdot n = 16 \cdot T\left(\frac{n}{16}\right) + 4c \cdot n \)

\[
\leq \ldots \leq 2^k \cdot T\left(\frac{n}{2^k}\right) + k \cdot c \cdot n
\]

\[
\leq 2^{\log n} \cdot T\left(\frac{n}{2^{\log n}}\right) + c \cdot n \log n
\]

\(< \) corresponds to recursion depth \( k \)

\( \text{shy - when } \frac{n}{2^k} = 1 \Rightarrow k = \log n \)
A recurrence

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

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

-> “substitution / induction”

\[
\begin{align*}
T(n) & \leq c \quad \text{for } n = 1 \\
T(n) & \leq 2T\left(\frac{n}{2}\right) + cn \quad \text{for } n > 1
\end{align*}
\]

guess a function \( g(n) \), show by induction that \( T(n) \leq g(n) \)

guess: \( g(n) = cn \log n + cn \)

**BASE CASE:** \( n = 1 \): need to show: \( T(1) \leq g(1) = 1 \log 1 = 0 \)

\( n = 2 \):

we know: \( T(2) \leq 2T(1) + c \cdot 2 \leq 4c \) \( \checkmark \)

**IND. CASE:** \( n > 2 \) need to show: \( T(n) \leq g(n) = cn \log n + cn \)

(assume a power of 2) we know: \( T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq 2 \cdot g\left(\frac{n}{2}\right) + c \cdot n = 2 \cdot \left[c \cdot \frac{n}{2} \log \frac{n}{2} + c \cdot \frac{n}{2}\right] + c \cdot n \\
\text{IH for } n/2: \ T\left(\frac{n}{2}\right) \leq g\left(\frac{n}{2}\right) \)
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
- add key
- change key value
More on sorting

Heap
- stored in an array – how to compute: Parent
- how to add a key? Left child

Right child
More on sorting

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

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

**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:
    - otherwise?
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. $H = \text{BUILD\_HEAP}(A)$
2. $n = A.\text{length}$
3. for $i=0$ to $n-1$ do
4. $A[i] = \text{EXTRACT\_MIN}(H)$

BUILD\_HEAP(A)
1. initially $H = \emptyset$
2. $n = A.\text{length}$
3. for $i=0$ to $n-1$ do
4. ADD($H$, $A[i]$)

EXTRACT\_MIN(H)
1. $min = H[0]$
2. $H.\text{length}--$
3. $H[0] = H[H.\text{length}]$
4. \text{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 datastructures
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 \( \Omega(n \log n) \).
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 $O(n \log n)$? 

for specific inputs
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.

**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 $B[x]$

Running time?