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

\[
\begin{align*}
A &= \{1, 7, 10, 12, 19\} \\
B &= \{2, 4, 5, 8, 11, 16, 22, 100\} \\
C &= \{1, 2, 4, 5, 7, 8, 10, 11, 12, 16, 19, 22, 100\}
\end{align*}
\]
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? $O(n+m)$
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. $A_s = \text{MERGESORT}(A,\text{middle})$
6. $B_s = \text{MERGESORT}(B,\text{n-middle})$
7. RETURN MERGE($A_s,B_s$)
Running times continued

**Sorting**

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

Output: \( X \) sorted in increasing order

**MergeSort**

\[
\text{MERGESORT}(X,n) \quad T(n)
\]

1. if \((n == 1)\) RETURN \( X \)
2. \( \text{middle} = \frac{n}{2} \) (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. \( As = \text{MERGESORT}(A, \text{middle}) \quad T(n/2) \)
6. \( Bs = \text{MERGESORT}(B, n-\text{middle}) \quad T(n/2) \)
7. RETURN MERGE(\(As, Bs\))
A recurrence

Running time of MergeSort:

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

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

\[ T(n) \leq c \]

\( n = 1 \)

\( \forall n \geq 2 \)

\( \text{for simplicity, assume } n \text{ is a power of 2} \)

\[ 
\begin{align*}
T(n) &\leq 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n \\
&\leq 2 \cdot \left( 2 \cdot T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2} \right) + c \cdot n = \\
&= 4 \cdot T\left(\frac{n}{4}\right) + c \cdot n + c \cdot n = 4 \cdot T\left(\frac{n}{4}\right) + 2cn \leq \\
&\leq 4 \cdot \left( 2 \cdot T\left(\frac{n}{8}\right) + c \cdot \frac{n}{4} \right) + 2cn = 8 T\left(\frac{n}{8}\right) + 3cn \leq \\
&\leq 2^k \cdot T\left(\frac{n}{2^k}\right) + k \cdot c \cdot n
\end{align*}
\]

\( \frac{n}{2^k} = 1 \) we use the 2nd ineq. in the box above (in the recurrence)

this happens for \( k = \log n \)

\( n = 2^k = 2^{\log n} \)

\[ \leq 2^k \cdot c + k \cdot c \cdot n = cn + c \cdot n \log n = O(n \log n) \]
A recurrence

Running time of MergeSort: \[T(n) \leq 2T\left(\frac{n}{2}\right) + cn\] \(\forall n \geq 2\)

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

-> “substitution / induction”

We will prove that \(T(n) \leq d \cdot n \cdot \log n\) where \(d\) is a constant.

**BASE CASE:** \(n = 1\)

**IND. CASE:** \(n > 2\)

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

IH: \(T(k) \leq d \cdot k \cdot \log k\) \(\forall k < n\)

we know: \(\frac{n}{2} = \log n\)

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

we want to show that \(T(n) \leq d \cdot n \cdot \log n\)

\(T(n) \leq 2T\left(\frac{n}{2}\right) + cn = 2T(1) + 2c \leq 2c + 2c = 4c = 0\)

\(T(n) \leq 4c = 2d\)

\(d \geq 2c\)

\(\therefore d \geq 2c\)

**problem:** \(c \neq 0\)

Solutions:

1. change base case to bigger number (corresponds to taking bigger \(n_0\))
2. change the eq.
More on sorting

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

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

we will do HeapSort

(also quicksort is expected
run-time $O(n \log n)$
but $O(n^2)$ in the worst case)

if can use only
comparisons, then need $O(n \log n)$ comparisons

if we have special input data,
then we might be able to run faster

but not for general input data
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:
  \[ \text{Parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor \]
  \[ \text{Left child}(i) = 2i + 1 \]
  \[ \text{Right child}(i) = 2i + 2 \]
- how to add a key?

Add new element to the end of the heap, and while it is smaller than its parent, swap them.
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 \((H[i] < H[\text{Parent}(i)])\) do
2. swap entries \(H[i]\) and \(H[\text{Parent}(i)]\)
3. \(i = \text{Parent}(i)\)

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

Heap
- stored in an array - how to compute:
  - if key decreased, then: heapify-up
  - otherwise?

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: heapify-up
  - otherwise?
More on sorting

Heap
- stored in an array - how to compute:

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

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)?
More on sorting

Heap
- running times:

\[ n = \#\text{elem. in heap} \]

\[ 2^{k-1} = 1 + 2 + 4 + \ldots + 2^{k-2} \leq n \leq 1 + 2 + 4 + 8 + \ldots + 2^{k-1} = 2^k - 1 \]

Use: priority queue - a datastructure that supports:
- extract-min \( O(\log n) \)
- add key \( O(\log n) \)
- change key value \( O(\log n) \)
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

Note (more efficient BUILD_HEAP):
A different implementation of BUILD_HEAP runs in time O(n).
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. \( \text{ADD}(H,A[i]) \)

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

Running time:
Related datastructures

- **Binary Search Trees**
  - Structure:
    
    ![Tree Diagram](image)

  - Operations:
    - Insert
    - Delete
    - Search

- **Balanced Trees**
  - Properties:
    - Each internal node has two non-empty subtrees
    - Height difference between subtrees is at most 1

  - Example of a balanced tree:
    
    ![Tree Diagram](image)

- **Red-Black Trees**
  - Balanced by rotations
  - Crucial operation: Rotation

  - Example of rotation:
    
    ![Tree Diagram](image)
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.

1. 173
2. 278
3. 123
4. 237
5. 635
6. 187
7. 187
8. 278
9. 187

1. 123
2. 273
3. 635
4. 273
5. 178
6. 278
7. 187
8. 635
9. 187

Curried: add to list in $O(1)$ time to get output in $O(n)$.
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? $O(dn)$ where $d =$ max # digits