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

\[
A = 1, 5, 6, 7, 10, 11, 15 \\
B = 2, 4, 8, 9, 12
\]
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 $X = \{x_1, x_2, ..., x_n\}$

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(\( A, \text{middle} \))
6. \( Bs = \) MERGESORT(\( B, n-\text{middle} \))
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($X, n$) $\leftarrow T(n)$
1. if ($n == 1$) RETURN $X$
2. 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. $A_s = \text{MERGESORT}(A, \text{middle})$ $\leftarrow T(\frac{n}{2})$
6. $B_s = \text{MERGESORT}(B, n-\text{middle})$ $\leftarrow T(\frac{n}{2})$
7. RETURN $\text{MERGE}(A_s, B_s)$ $\leftarrow O(n)$

Running time?

$T(n) = \# \text{skips for input of size } n$

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

For $n \geq 1$

$T(1) = O(1)$

For $n \neq 1$
A recurrence

Running time of MergeSort: $T(n)$

How to bound $T(n)$?

$->$ “unrolling the recurrence”

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

\[
T(n) = O(1) \quad \text{for } n = 1
\]

Rewrite:

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

\[
T(n) \leq c \quad \text{for } n \leq 1
\]

Then

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

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

\[
\leq ... = 2^k T\left(\frac{n}{2^k}\right) + kcn \leq nc + \log n \cdot c \cdot n = O(n \log n)
\]

Will stop when $\frac{n}{2^k} \leq 1$, i.e., $k \geq \log n$

(suppose $k = \log n$)

Running time:

\[
\text{Run depth: } (c+d)n = O(n \log n)
\]

\[
\text{Depth } \leq \log n
\]
A recurrence

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

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

\( \rightarrow \) “substitution / induction”

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

\[
T(n) \leq c \quad \text{for } n \leq 1
\]

Guess that \( T(n) \leq d \cdot n \cdot \log n \) for every \( n \geq 2 \)

We induction to verify the guess

**BASE CASE:** \( n = 2 \)

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

Want \( T(2) \leq d \cdot 2 \cdot \log 2 = 2d \)

Holds if \( 4c \leq 2d \)

i.e. \( d \geq 2c \)

**IND. CASE:** For simplicity, assume \( n \) is a power of \( 2 > 2 \)

Want to show \( T(n) \leq d \cdot n \cdot \log n \), assuming (inductive hypothesis): \( T\left(\frac{n}{2}\right) \leq d \cdot \frac{n}{2} \cdot \log \frac{n}{2} \)

We know:

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

Want \( \leq dn \log n \)
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
  - implement using linked list: $O(n)$
  - sorted linked list: $O(1)$
- add key
  - $O(1)$
- change key value
  - $O(1)$
More on sorting

Heap
- stored in an array - how to compute:

- how to add a key?

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

add key 2
More on sorting

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

- 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)

running time $O(\log n)$ i.e. $O(\text{depth})$ and depth $= \lceil \log (n+1) \rceil$
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: \texttt{HEAPIFY-UP}
  - otherwise?

idea:
  swap with the smaller child
  (assuming its key < newkey)
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

running time: O(log n)
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).
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

Balanced binary search tree:
1) find an element: $O(\log n)$
2) adding/removed an element: $O(\log n)$

Adding 9:

Rebalancing:
(right rotation at 5)
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 $O(n \log n)$?

*if the input is guaranteed to be "nice"*
Sorting faster than $O(n \log n)$?

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

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

```
5 2 7
3 7 2
2 5 7
3 5 5
7 2 2
3 3 3

1 372 \rightarrow 722
2 333
3 355
4 527 \rightarrow 257
5 722
```

Sort w. 2nd to last digit

```
2 722 \rightarrow 527
3 333
5 355 \rightarrow 257
```

3rd to last digit

```
2 372
3 355 \rightarrow 372
5 527
7 722
```

Takes $O(n)$ time

(assuming addition to a linked list is $O(1)$)

Running time: $O(d \cdot n)$ where $d$ = # digits of the largest number
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?