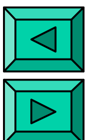
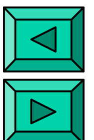
Worst-Case Performance

- **Upper bounds**: simple to obtain
- **Lower bounds**: ***a difficult matter***...
- Worst case data may be unlikely to be met in practice
- Unknown "Big-Oh" constants c and n_0 may not be small
- Inputs in practice lead to much lower running times
- ***Example:*** the most popular fast sorting algorithm, **QuickSort**, has $\mathrm{O}(n^2)$ running time in the worst case but in practice the time is $\mathrm{O}(n \log n)$



Average-Case Performance

- Estimate average time for each operation
- Estimate frequencies of operations
- May be a difficult challenge... (take **COMPSCI.320** for details)
- May be no natural “average” input at all
- May be hard to estimate (average time for each operation depends on data)

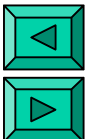


Recurrent Algorithms

Divide-and-conquer principle:

- divide a large problem into smaller ones and recursively solve each sub-problem, then
- combine solutions of sub-problems to solve the original problem

Running time: by a ***recurrence relation*** combining the size and number of sub-problems and the cost of dividing the problem into sub-problems



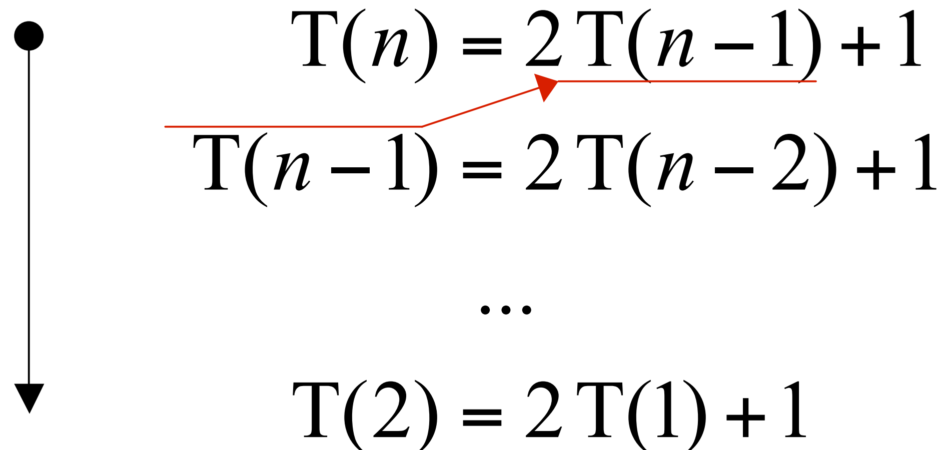
The University of Auckland

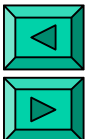
"Telescoping" a Recurrence

- Recurrence relation and its base condition (i.e., the difference equation and initial condition):

$$T(n) = 2 \cdot T(n-1) + 1; \; T(1) = 1$$

- Closed (explicit) form for $T(n)$ by "telescoping":





"Telescoping" ≡ Substitution

$$T(n) = 2\,T(n-1) + 1$$

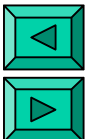
$$2\,T(n-1) = 2^2\,T(n-2) + 2$$

$$2^2\,T(n-2) = 2^3\,T(n-3) + 2^2$$

$$\dots$$

$$2^{n-1}\,T(2) = 2^n\,T(1) + 2^{n-1}$$

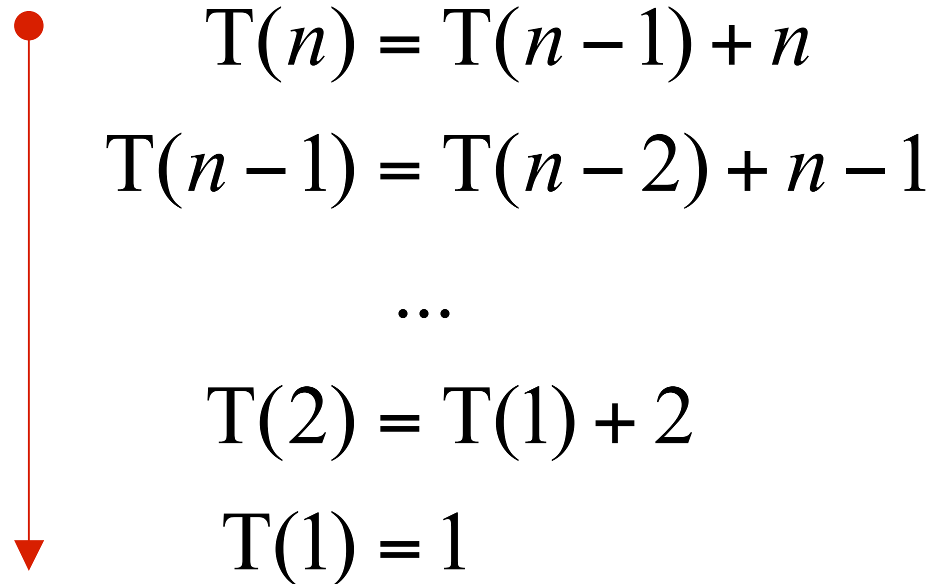
$$T(n) = 1 + 2 + 2^2 + \ldots + 2^{n-1} + 2^n = 2^{n+1} - 1$$



Basic Recurrence: 1

$$\mathrm{T}(1) = 1, \mathrm{T}(n) = \mathrm{T}(n-1) + n \quad \Leftrightarrow \quad \mathrm{T}(n) = \frac{n(n+1)}{2}$$

- Closed (explicit) form by “telescoping”:



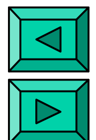
$$\mathrm{T}(n) = \mathrm{T}(n-1) + n$$

$$\mathrm{T}(n-1) = \mathrm{T}(n-2) + n - 1$$

$$\ldots$$

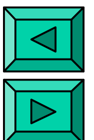
$$\mathrm{T}(2) = \mathrm{T}(1) + 2$$

$$\mathrm{T}(1) = 1$$



1: Explicit Expression for T(*n*)

$$
\begin{aligned}
\mathrm{T}(n) &= \mathrm{T}(n-1) + n \\
&= \overline{\mathrm{T}(n-2) + (n-1)} + n \\
&= \overline{\mathrm{T}(n-3) + (n-2)} + (n-1) + n \\
&\ldots \\
&= \overline{\mathrm{T}(2) + 3} + \ldots + (n-2) + (n-1) + n \\
&= \overline{\mathrm{T}(1) + 2} + \ldots + (n-2) + (n-1) + n \\
&= \overline{1} + 2 + \ldots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}
\end{aligned}
$$



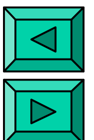
Guessing to Solve a Recurrence

- Infer (guess) a hypothetic solution $T(n)$; $n \geq 0$ from a sequence of numbers $T(0)$, $T(1)$, $T(2)$, …, obtained from the recurrence relation
- Prove $T(n)$ by math induction:

  Base condition: T holds for $n = n_{base}$,e.g. $T(0)$ or $T(1)$

  Induction hypothesis: prove that for every $n > n_{base}$, if T holds for $n - 1$, then T holds for n

  Strong induction: if T holds for $n_{base}, \dots, n - 1$, then…

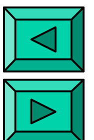


2: Explicit Expression for T(*n*)

- $T(1) = 1$; $T(2) = 1 + 2 = 3$; $T(3) = 3 + 3 = 6$; $T(4) = 6 + 4 = 10 \Rightarrow$ Hypothesis: $T(n) = \frac{n(n+1)}{2}$
- Base condition holds: $T(1) = 1 \cdot 2 / 2 = 1$
- If the hypothetic formula $T(n)$ holds for $n - 1$, then it holds also for n:

$$T(n) = T(n-1) + n = \frac{(n-1)n}{2} + n = \frac{n(n+1)}{2}$$

- Thus, the expression for $T(n)$ holds for all $n > 1$



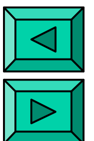
Basic Recurrence: 2

- **Repeated halving principle**: halve the input in one step:

$$T(1) = 0, T(n) = T(n/2) + 1 \quad \Leftrightarrow \quad T(n) \cong \log_2 n$$

- "Telescoping" (for $n = 2^m$):

$$T(2^m) = T(2^{m-1}) + 1$$
$$T(2^{m-1}) = T(2^{m-2}) + 1$$
$$\ldots$$
$$T(2^2) = T(2^1) + 1$$
$$T(2^1) = T(2^0) + 1$$
$$T(2^0) = 0$$



3: Explicit Expression for T(*n*)

$$T(2^m) = T(2^{m-1}) + 1$$

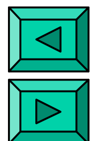
$$= \overline{T(2^{m-2}) + 1} + 1$$

$$\dots$$

$$= \overline{T(2^1) + 1} + 1 + \dots + 1$$

$$= \overline{T(2^0) + 1} + 1 + \dots + 1 + 1$$

$$T(2^m) = m \quad \Rightarrow \quad T(n) = \log_2 n$$



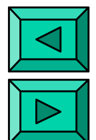
Basic Recurrence: 3

- Scan and halve the input:

$$T(1) = 1, T(n) = T(n/2) + n \quad \Leftrightarrow \quad T(n) \cong 2n$$

- "Telescoping" (for $n = 2^m$):

$$T(2^m) = T(2^{m-1}) + 2^m$$
$$T(2^{m-1}) = T(2^{m-2}) + 2^{m-1}$$
$$\ldots$$
$$T(2^2) = T(2^1) + 2^2$$
$$T(2^1) = T(2^0) + 2^1$$
$$T(2^0) = 1$$



4: Explicit Expression for T(*n*)

$$T(2^m) = T(2^{m-1}) + 2^m$$

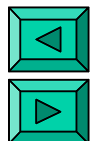
$$= \overline{T(2^{m-2}) + 2^{m-1}} + 2^m$$

$$\dots$$

$$= \overline{T(2^1) + 2^2} + \ldots + 2^{m-1} + 2^m$$

$$= \overline{T(2^0) + 2^1} + 2^2 + \ldots + 2^{m-1} + 2^m$$

$$T(2^m) = 2^{m+1} - 1 \quad \Rightarrow \quad T(n) \cong 2n$$



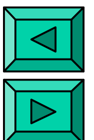
Basic Recurrence: 4

- **"Divide-and-conquer" prototype:**

$$T(1) = 0, T(n) = 2T(n/2) + n \quad \Leftrightarrow \quad T(n) \cong n \log_2 n$$

- "Telescoping": $\dfrac{T(n)}{n} = \dfrac{T(n/2)}{n/2} + 1; \quad T(1) = 0$

- For $n = 2^m \rightarrow \dfrac{T(2^m)}{2^m} = \dfrac{T(2^{m-1})}{2^{m-1}} + 1$



4: Explicit Expression for T(*n*)

$$T(2^m)/2^m = T(2^{m-1})/2^{m-1} + 1$$

$$= \overline{T(2^{m-2})/2^{m-2} + 1} + 1$$

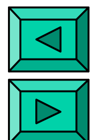
$$\ldots$$

$$= \overline{T(2^1)/2^1 + 1} + \ldots + 1 + 1$$

$$= \overline{T(2^0)/2^0 + 1} + 1 + \ldots + 1 + 1$$

$$= 0 + 1 + \ldots + 1 = m$$

$$T(2^m) = m \cdot 2^m \quad \Rightarrow \quad T(n) = n \log_2 n$$

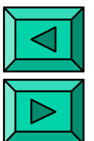


General "Divide-and-Conquer"

Theorem: The recurrence $\mathrm{T}(n) = a\,\mathrm{T}(n/b) + cn^k$; $\mathrm{T}(1) = c$ with integer constants $a \geq 1$ and $b \geq 2$ and positive constants c and k has the solution:

$$\mathrm{T}(n) = \begin{cases} O(n^{\log_b a}) & \text{if} \quad b^k < a \\ O(n^k \log n) & \text{if} \quad b^k = a \\ O(n^k) & \text{if} \quad b^k > a \end{cases}$$

Proof by telescoping: $n = b^m \Rightarrow \mathrm{T}(b^m) = a\,\mathrm{T}(b^{m-1}) + cb^{mk}$



General “Divide-and-Conquer”

Telescoping:

$$T(b^m) = aT(b^{m-1}) + cb^{mk}$$

$$aT(b^{m-1}) = a^2T(b^{m-2}) + acb^{(m-1)k}$$

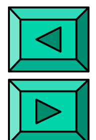
$$a^2T(b^{m-2}) = a^3T(b^{m-3}) + a^2cb^{(m-2)k}$$

$$\ldots = \ldots$$

$$a^{m-1}T(b) = a^mT(1) + a^{m-1}cb^k$$

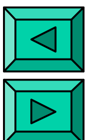
$$\mathrm{T}(b^m) = a^mc + a^{m-1}cb^k + \ldots + acb^{(m-1)k} + cb^{mk}$$

$$= c\sum_{t=0}^{m} a^{m-t}b^{tk} = ca^m \sum_{t=0}^{m}\left(\tfrac{b^k}{a}\right)^t$$



Capabilities and Limitations

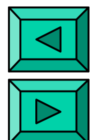
- Rough complexity analysis cannot result immediately in an efficient practical program but may help in predicting the empirical running time of the program
- "Big-Oh" analysis is unsuitable for small input and hides the constants c and n_0 crucial for practical applications
- "Big-Oh" analysis is unsuitable if costs of access to input data items vary and if there is lack of sufficient memory
- But complexity analysis provides ideas how to develop new efficient methods



Data Sorting

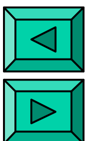
- **Order relation:** places each pair a, b of ***countable*** items in a fixed order denoted as (a,b) or **<a,b>**
- **Order notation:** $a \leq b$ (*less than or equal to*)
- **Countable item:** labelled by a specific ***integer key***
- **Comparable objects in Java:** if an object can be ***less than***, ***equal to***, or ***greater than*** another object:

object1.compareTo(object2) **<0, =0, >0**



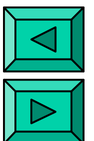
Order of Data Items

- **Numerical order** - by value:
  $5 \le 5 \le 6.45 \le 22.79 \le \ldots \le 1056.32$
- **Alphabetical order** - by position in an alphabet:
  a ≤ b ≤ c ≤ d ≤ … ≤ z
  The order depends on the alphabet used: look into any bilingual dictionary...
- **Lexicographic order** - by first differing element:
  5456 ≤ 5457 ≤ 5500 ≤ 6100 ≤ …
  pork ≤ ward ≤ word ≤ work ≤ …



Properties of an Ordering

- A relation on an array $\mathbf{A} = \{a, b, c, \ldots\}$ is:
  - ***reflexive:*** if $a \leq a$
  - ***transitive:*** if $a \leq b$ and $b \leq$ c, then $a \leq c$
  - ***symmetric:*** if $a \leq b$ and $b \leq$ a, then $a = b$
- **Linear order** if for any pair of elements a and b either $a \leq b$ or $b \leq a$: $a \leq b \leq c \leq \ldots$
- **Partial order** if there are incomparable elements



Insertion Sort

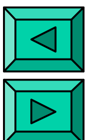
- Splits an array into two parts, ordered & unordered
- Sequentially contracts the unordered part, one element per stage:

ordered part — unordered part

$a_0, \ldots, a_{i-1}$ | $a_i, \ldots, a_{n-1}$

- Stage $i = 1, \ldots, n-1$:

$n - i$ unordered and i ordered elements

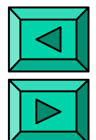


The University of Auckland

Example of Insertion Sort

- N_c - number of comparisons per insertion
- N_m - number of moves per insertion

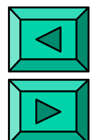
13	18	35	44	*15*	*10*	*20*	N_c	N_m
			15	44			$<$	$\rightarrow$
		15	35				$<$	$\rightarrow$
	15	18					$<$	$\rightarrow$
15							$\geq$	
13	15	18	35	44	*10*	*20*	4	3



The University of Auckland

Example of Insertion (Cont)

13	15	18	35	44	*10*	*20*	N_c	N_m
				10	44		$<$	$\rightarrow$
			10	35			$<$	$\rightarrow$
		10	18				$<$	$\rightarrow$
	10	15					$<$	$\rightarrow$
10	13						$<$	$\rightarrow$
10	13	15	18	35	44	*20*	5	5

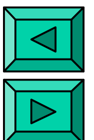


Implementation of Insertion Sort

begin InsertionSort (integer array a of size n)

1. for $i \leftarrow 1$ to $n - 1$ do
2. $temp \leftarrow a[\,i\,]; \quad k \leftarrow i - 1$
3. while $k \geq 0$ and $temp < a[k]$ do
4. $a[\,k+1\,] \leftarrow a[\,k\,]; \quad k \leftarrow k - 1$
5. end while
6. $a[\,k+1\,] \leftarrow temp$
7. end for

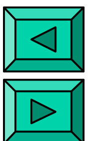
end InsertionSort



Average Complexity at Stage *i*

- $i + 1$ positions to place a next item: $0\ 1\ 2\ \ldots\ i\ \text{-}1\ i$
- $i - j + 1$ comparisons and $i - j$ moves for each position $j = i, i{-}1, \ldots, 1$
- i comparisons and i moves for position $j = 0$
- Average number of comparisons:

$$E_i = \frac{1+2+\ldots+i+i}{i+1} = \frac{i}{2} + \frac{i}{i+1}$$



Total Average Complexity

- $n - 1$ stages for n input items: the total average number of comparisons:

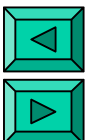
$$E = E_1 + E_2 + \ldots + E_{n-1}$$

$$= \tfrac{1}{2}(1 + 2 + \ldots + (n-1)) + \left(\tfrac{1}{2} + \tfrac{2}{3} + \ldots + \tfrac{n-1}{n}\right)$$

$$= \frac{n^2}{4} + \frac{3n}{4} - \left(1 + \tfrac{1}{2} + \tfrac{1}{3} + \ldots + \tfrac{1}{n}\right)$$

$$= \frac{n^2}{4} + \frac{3n}{4} - H_n$$

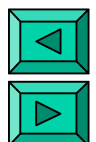
$$H_n \cong \ln n + 0.577, \;\; E \cong \Theta(n^2)$$



Analysis of Inversions

- An **inversion** in an array $\mathbf{A} = [a_1, a_2, \ldots, a_n]$ is any ordered pair of positions (i, j) such that $i < j$ but $a_i > a_j$: e.g., $[\ldots, 2, \ldots, 1]$ or $[100, \ldots, 35, \ldots]$

A	# invers.	$\mathbf{A}_{\text{reverse}}$	# invers.	Total #
3,2,5	1	5,2,3	2	3
3,2,5,1	4	1,5,2,3	2	6
1,2,3,4,7	0	7,4,3,2,1	10	10

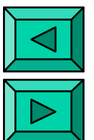


Analysis of Inversions

- Total number of inversions both in an arbitrary array $\mathbf{A}$ and its reverse $\mathbf{A}_{\text{reverse}}$ is equal to the total number of the ordered pairs ($i < j$):

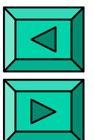
$$\binom{n}{2} = \frac{(n-1)\cdot n}{2}$$

- A sorted array has no inversions
- A reverse sorted array has $\frac{(n-1)n}{2}$ inversions



Analysis of Inversions

- Exactly one inversion is removed by swapping two neighbours $a_{i-1} > a_i$
- An array with k inversions results in $\mathrm{O}(n + k)$ running time of InsertionSort
- Worst-case time: $c\frac{n^2}{2}$, or $\mathrm{O}(n^2)$
- Average-case time: $c\frac{n^2}{4}$, or $\mathrm{O}(n^2)$



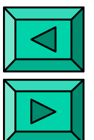
InsertionSort: A Critique

- Insertion sort is efficient if the input is almost sorted.
- Insertion sort is inefficient, on average, because it moves values just one position at a time.
- Idea:

  arrange the data sequence in a

  two-dimensional array

  sort the columns of the array



An Example

Start with the sequence

3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2

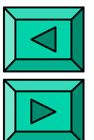
to be sorted.

First, arrange it in an array with 7 columns (left), then the columns are sorted

3 7 9 0 5 1 6   3 3 2 0 5 1 5

8 4 2 0 6 1 5   → 7 4 4 0 6 1 6

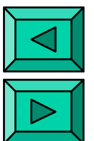
7 3 4 9 8 2   8 7 9 9 8 2



An Example (Cont)

In the next step, the sequence is arranged in 3 columns, which are again sorted:

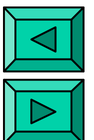
3 3 2		0 0 1
0 5 1		1 2 2
5 7 4		3 3 4
4 0 6	→	4 5 6
1 6 8		5 6 8
7 9 9		7 7 9
8 2		8 9



An Example (Cont)

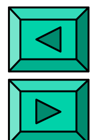
The sequence is almost completely sorted. When arranging it in one column in the last step, it is only a 6, an 8 and a 9 that have to move a little bit to their correct position.

In reality, the data sequence is not arranged in a two-dimensional array, but held in a one-dimensional array that is indexed appropriately.



ShellSort

- **ShellSort** eliminates more than just one inversion between the neighbours per exchange! (insertion sort eliminates one inversion per exchange)
- D.Shell (1959): compare first the keys at a distance of gap$_T$, then of gap$_{T-1}$ < gap$_T$, and so on until of gap$_1$=1
- After a stage with gap$_t$, all elements spaced gap$_t$ apart are sorted; it can be proven that any gap$_t$-sorted array remains gap$_t$-sorted after being then gap$_{t-1}$-sorted

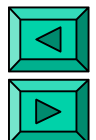


Implementation of ShellSort

begin ShellSort (integer array a of size n)

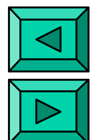
1. for gap ← $\lfloor n/2 \rfloor$ while gap > 0
   step gap ← (if gap = 2 then 1 else $\lfloor$gap$/2.2\rfloor$) do
2. for i ← gap while $i < n$ step $i \leftarrow i + 1$ do
3. $s_{tmp} \leftarrow a[i];\ \ k \leftarrow i$
4. while($k \geq$ gap and $s_{tmp} < a[\,k$ − gap] do
5. $a[\,k\,] \leftarrow a[\,k$ − gap]; $k \leftarrow k$ − gap
6. end while
7. $a[\,k\,] \leftarrow s_{tmp}$; 8. end for; 9. end for

end ShellSort



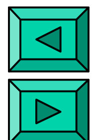
Example of ShellSort: step 1

gap	*i*:C:M	Data to be sorted									
		25	8	2	91	70	50	20	31	15	65
5	5:1:0	25					50				
	6:1:0		8					20			
	7:1:0			2					31		
	8:1:1				**15**					**91**	
	9:1:1					**65**					**70**
		25	8	2	15	65	50	20	31	91	70



Example of ShellSort: step 2

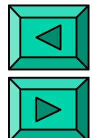
gap	*i*:C:M	25	8	2	15	65	50	20	31	91	70
2	2:1:1	**2**		**25**							
	3:1:0		8		15						
	4:1:0			25		65					
	5:1:0				15		50				
	6:3:2	2		**20**		**25**		**65**			
	7:2:1				15		**31**		**50**		
	8:1:0							65		91	
	9:1:0								50		70



The University of Auckland

Example of ShellSort: step 3

gap	*i*:C:M	2	8	20	15	25	31	65	50	91	70
1	1:1:0	2	8								
	2:1:0		8	20							
	3:2:1		8	**15**	**20**						
	4:1:0				20	25					
	5:1:0					25	31				
	6:1:0						31	65			
	7:2:1						31	**50**	**65**		
	8:1:0								65	91	
	9:2:1								65	**70**	**91**

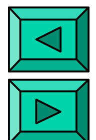


Time complexity of ShellSort

- Heavily depends on gap sequences
- Shell's sequence: $^{n}/_{2}, {}^{n}/_{4}, \ldots, 1$:

  $O(n^2)$ worst; $O(n^{1.5})$ average
- "Odd gaps only" (if even: $^{gap}/_{2} + 1$):

  $O(n^{1.5})$ worst; $O(n^{1.25})$ average
- Heuristic sequence: $^{gap}/_{2.2}$: better than $O(n^{1.25})$
- A very simple algorithm with an extremely complex analysis

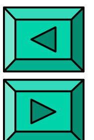


A Lower Bound for Sequential Sorting

Let L = $x_1\ x_2\ \dots\ x_n$ be a list to be sequentially sorted.

Four basic facts:

1. any comparison changes L into L' or L''
2. the process results in a binary tree whose terminal nodes correspond to potential sorted versions of L
3. any sorting corresponds to a permutation of L, so the final tree has $n!$ terminal nodes
4. the depth of a binary tree with m terminal nodes is at least $\lceil m \rceil$

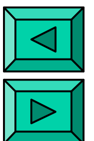


A Lower Bound for Sequential Sorting (Cont)

Hence:

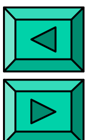
$$D \geq \lceil \log n! \rceil > \log n(n-1)(n-2)\ldots n/2$$

$$> \log(n/2)^{n/2} = \frac{n}{2}\log\frac{n}{2} = 0(n\log n)$$



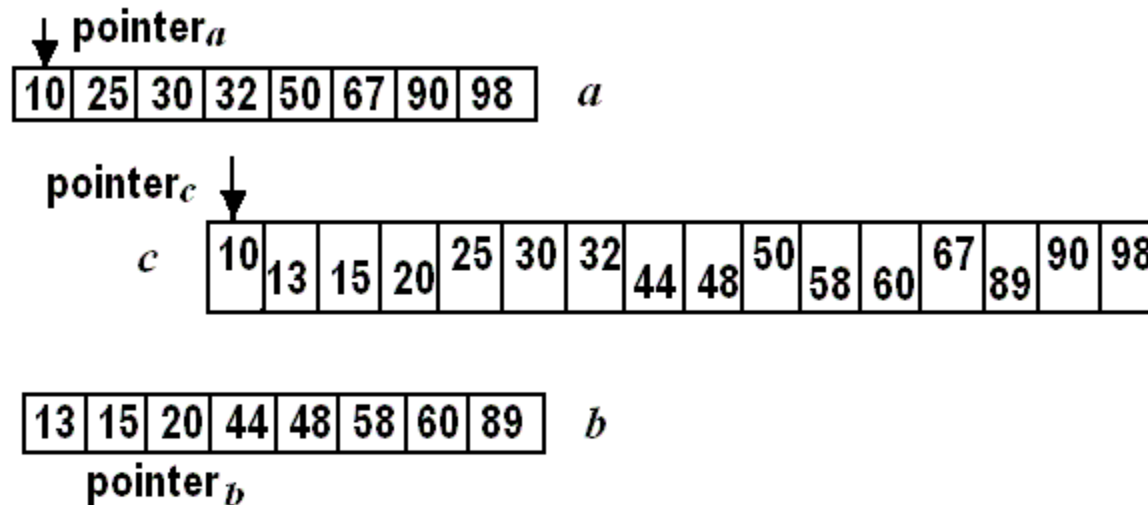
Algorithm MergeSort

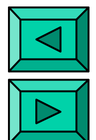
- **John von Neumann** (*1945!*): a recursive divide-and-conquer approach
- Three basic steps:
  - if the number of items is 0 or 1, return
  - recursively sort the first and the second halves separately
  - merge two presorted halves into a sorted array
- Linear time merging $O(n)$ yields MergeSort time complexity $O(n \log n)$



O(n) Merge of Sorted Arrays

if a[pointer$_a$] < b[pointer$_b$] then c[pointer$_c$] $\leftarrow$ a[pointer$_a$];
pointer$_a \leftarrow$ pointer$_a + 1$; pointer$_c \leftarrow$ pointer$_c + 1$
else c[pointer$_c$] $\leftarrow$ b[pointer$_b$];
pointer$_b \leftarrow$ pointer$_b + 1$; pointer$_c \leftarrow$ pointer$_c + 1$





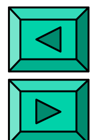
Structure of MergeSort

begin MergeSort (an integer array a of size n)

1. Allocate a temporary array tmp of size n
2. **RecursiveMergeSort**(a, tmp, 0, $n-1$)

end MergeSort

Temporary array: to merge each successive pair of ordered sub-arrays a[left], …, a[centre] and a[centre+1, …, a[right] and copy the merged array back to a[left], …, a[right]

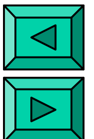


Recursive MergeSort

begin RecursiveMergeSort (an integer array a of size n;
a temporary array tmp of size n; range: left, right)

- if left < right then
- centre ← $\lfloor (\text{left} + \text{right}) / 2 \rfloor$
- **RecursiveMergeSort**(a, tmp, left, centre);
- **RecursiveMergeSort**(a, tmp, centre + 1, right);
- **Merge**(a, tmp, left, centre + 1, right);
- end if

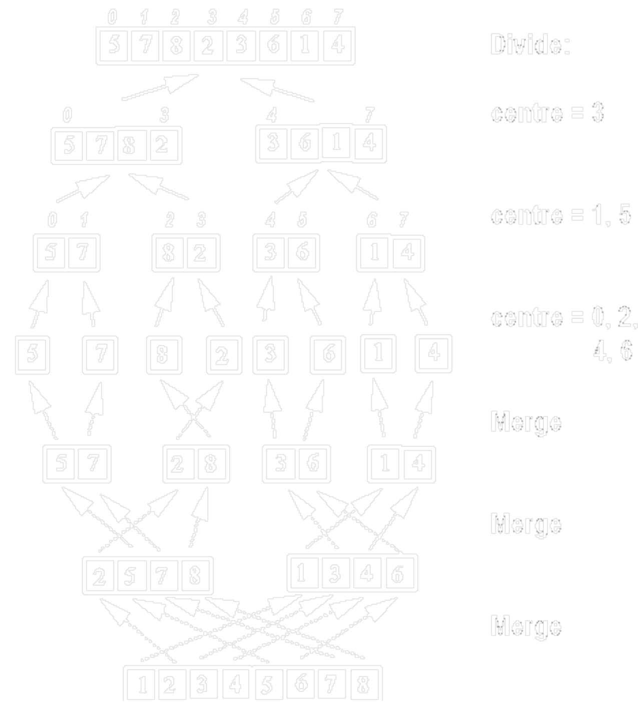
end RecursiveMergeSort

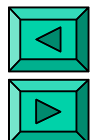


How MergeSort works

$2n$ comparisons for random data

n comparisons for sorted/reverse data





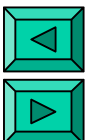
Analysis of MergeSort

$O(n \log n)$ best-, average-, and worst-casecomplexity because the merging is always linear

Extra O(n) temporary array for merging data

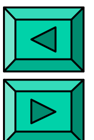
Extra work for copying to the temporary array and back

- Useful only for external sorting
- For internal sorting: QuickSort and HeapSort are much better



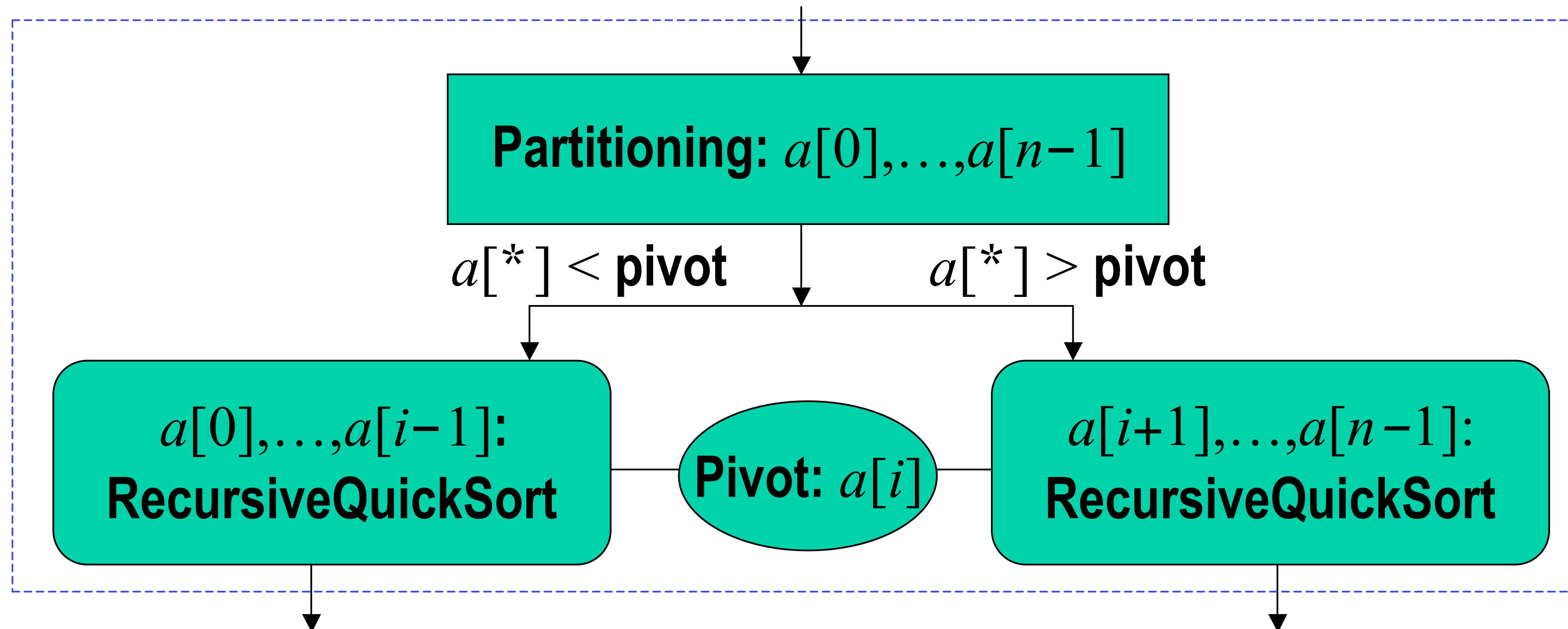
Algorithm QuickSort

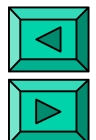
- C. A. R. Hoare (1961): the divide-and-conquer approach
- Four basic steps:
  - If $n = 0$ or 1, return
  - Pick a **pivot** item
  - Partition the remaining items into the left and right groups with the items that are smaller or greater than the pivot, respectively
  - Return the **QuickSort** result for the left group, followed by the pivot, followed by the **QuickSort** result for the right group



Recursive QuickSort

- $T(n) = c \cdot n$ (pivot positioning) + $T(i) + T(n - 1 - i)$

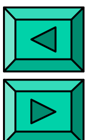




Analysis of QuickSort: the Worst Case $O(n^2)$

- If the pivot happens to be the largest (or smallest) item, then one group is always empty and the second group contains all the items but the pivot
- Time for partitioning an array: $c \cdot n$
- Running time for sorting: $T(n) = T(n-1) + c \cdot n$
- "Telescoping" (recall the basic recurrences):

$$T(n) = c \cdot \frac{n(n+1)}{2}$$



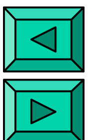
Analysis of QuickSort: the Average Case $O(n \log n)$

- The left and right groups contain i and $n - 1 - i$ items, respectively; $i = 0, \ldots, n - 1$
- Time for partitioning an array: $c{\cdot}n$
- Average running time for sorting:

$$T(n) = \tfrac{2}{n}\big(T(0) + \ldots + T(n-2) + T(n-1)\big) + cn, \text{ or}$$

$$nT(n) = 2\big(T(0) + \ldots + T(n-2) + T(n-1)\big) + cn^2$$

$$(n-1)T(n-1) = 2\big(T(0) + \ldots + T(n-2)\big) \qquad + c(n-1)^2$$



Analysis of QuickSort: the Average Case O(n log n)

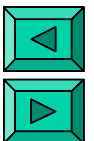
$$nT(n) - (n-1)T(n-1) \rightarrow nT(n) = (n+1)T(n-1) + 2cn-c$$

"Telescoping": $\frac{T(n)}{n+1} \cong \frac{T(n-1)}{n} + \frac{2c}{n+1}$

Explicit form : $\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n+1}\right)$

$$\approx 2cH_{n+1} \approx C \log n$$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n + 0.577$



Analysis of QuickSort: Choice of Pivot

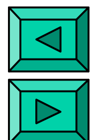
- Never use the first a[low] or the last a[high] item!
- A reasonable choice is the middle item:

$$a\left[\text{middle} = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor\right]$$

where $\lfloor z \rfloor$ is an integer "floor" of the real value z

- **A good choice is the median of three:**

a[low], a[middle], a[high]

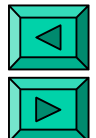


The University of Auckland

Pivot Positioning in QuickSort:

low=0 , middle=4, high=9

Data to be sorted										Description		
0	1	2	3	4	5	6	7	8	9	←Indices		
25	**8**	**2**	**91**	*70*	**50**	**20**	**31**	**15**	*65*	Initial array ***a***		
25	**8**	**2**	**91**	**65**	**50**	**20**	**31**	**15**	*70*	i=**MedianOfThree**($\boldsymbol{a}$,low,high];		
25	**8**	**2**	**91**	**15**	**50**	**20**	**31**	**65**	*70*	$p=a[i]$; **swap**(i, a[high−1])		
25	**8**	**2**	**91**	**15**	**50**	**20**	**31**	**65**	*70*	i	j	Condition
	8						**31**			1	7	$a[i]< p> a[j]$; i++
		2					**31**			2	7	$a[i]< p> a[j]$; i++

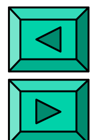


The University of Auckland

Pivot Positioning in QuickSort:

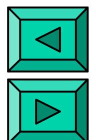
low=0 , middle=4, high=9

25	8	2	91	15	50	20	31	65	*70*	i	j	Condition
			91 31				31 91	65		3	7	$a[i] \geq p > a[j]$; swap; i++; j−−
				15		20		65		4	6	$a[i] < p > a[j]$; i++
					50	20		65		5	6	$a[i] < p > a[j]$; i++
						20		65		6	6	$a[i] < p > a[j]$; i++
								65		7	6	$i > j$; **break**
25	8	2	31	15	50	20	65	91	70	**swap**$(a[i], p = a[\text{high}-1])$		

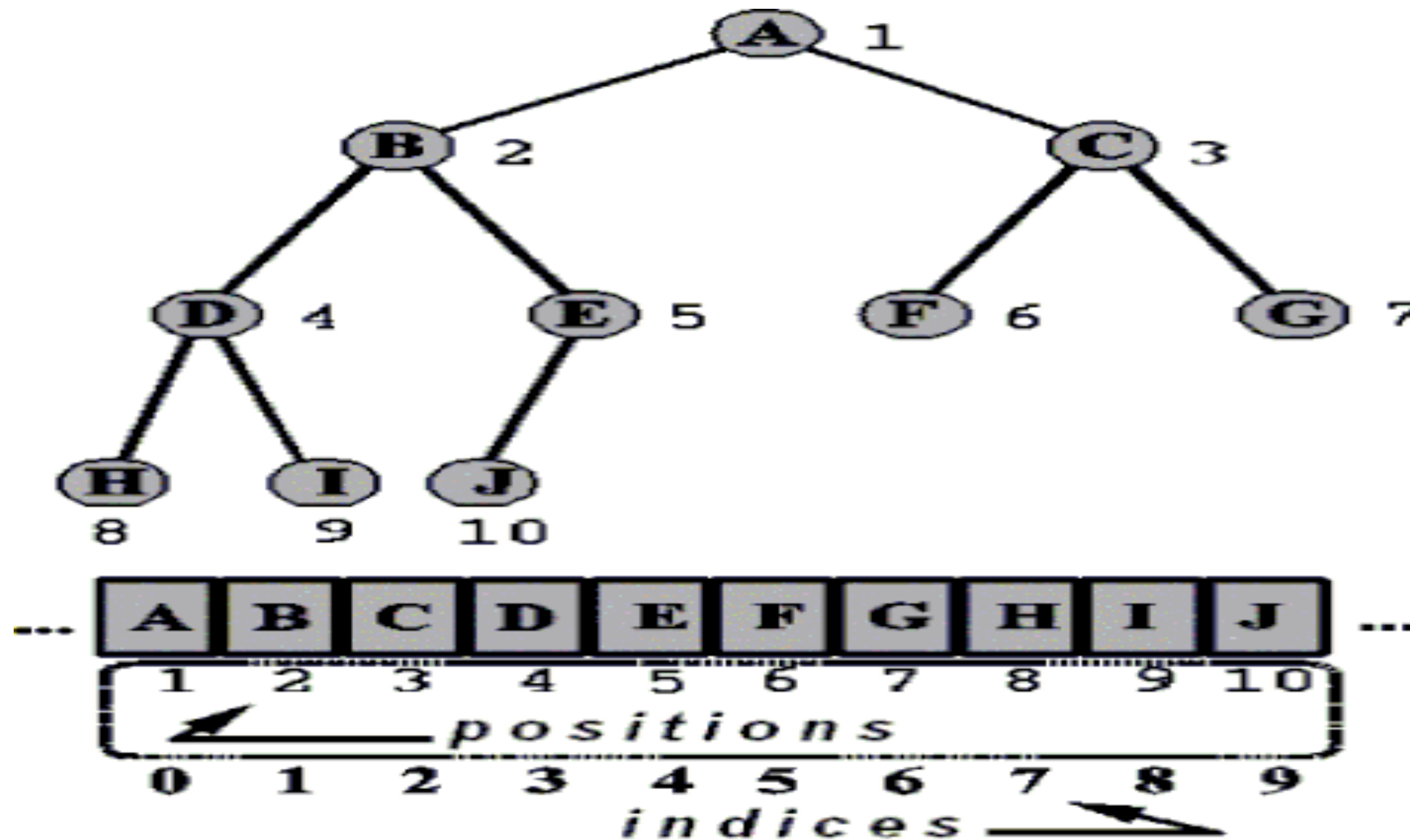


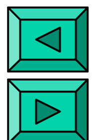
Algorithm HeapSort

- J. W. J. Williams (1964): a special binary tree called **heap** to obtain an $O(n \log n)$ worst-case sorting
- Basic steps:
  - Convert an array into a heap in linear time $O(n)$
  - Sort the heap in $O(n \log n)$ time by deleting n times the maximum item because each deletion takes the logarithmic time $O(\log n)$



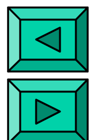
Binary Tree: Linear Array Representation





Complete Binary Tree

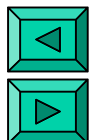
- A complete binary tree of the height h contains between 2^h and $2^{h+1}-1$ nodes
- A complete binary tree with the n nodes has the height $\lfloor \log_2 n \rfloor$
- Node positions are specified by the level-order traversal (the root position is 1)
- If the node is in the position p then:
  - the parent node is in the position $\lfloor p/2 \rfloor$
  - the left child is in the position $2p$
  - the right child is in the position $2p + 1$



Binary Heap

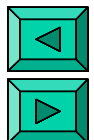
- A heap consists of a complete binary tree of height h with numerical keys in the nodes
- **The defining feature of a heap**:

  the key of each parent node is **greater than** or **equal to** the key of any child node
- The root of the heap has the maximum key



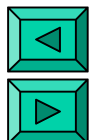
Binary Tree: Linear Array Representation

...	91	65	70	31	8	50	25	20	15	2	...
	1	2	3	4	5	6	7	8	9	10	



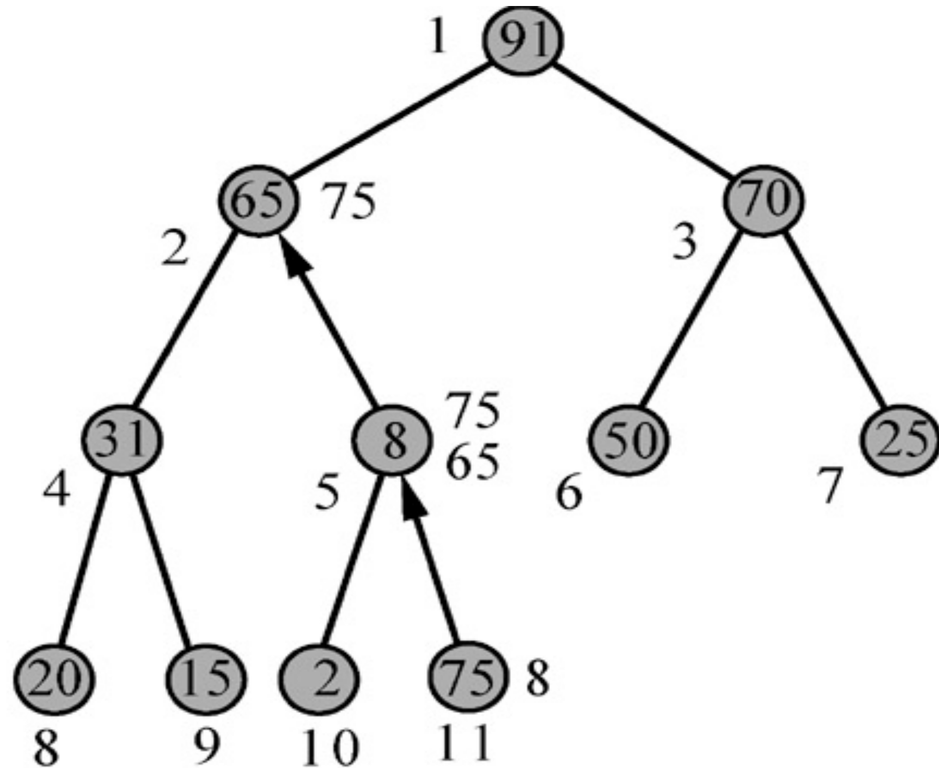
Binary Heap: Insert a New Key

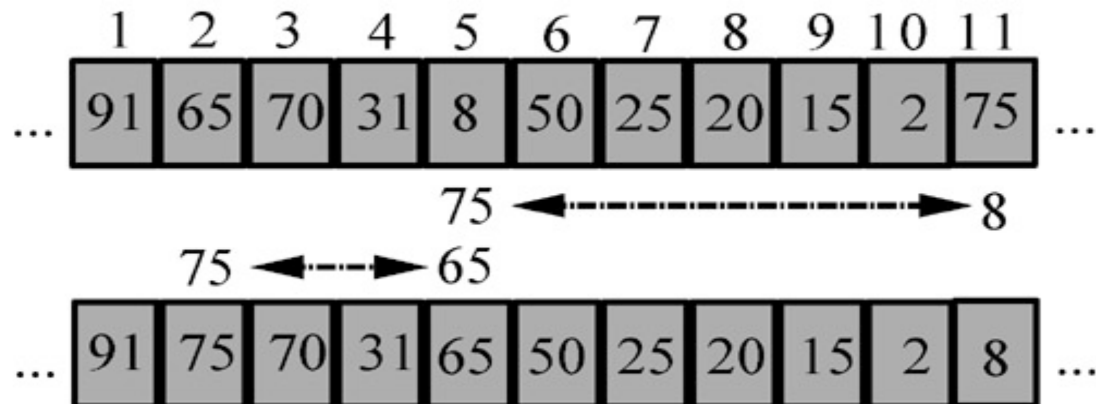
- Heap of k keys → heap of $k+1$ keys
- Logarithmic time $O(\log k)$ to insert a new key:
  - Create a new leaf position $k+1$ in the heap
  - **Bubble** (or **percolate**) the new key up by swapping it with the parent if the latter one is smaller than the new key

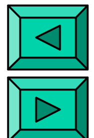


The University of Auckland

Binary Heap: An Example of Inserting a Key

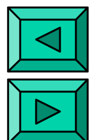




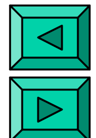


Binary Heap: Delete the Maximum Key

- Heap of k keys → heap of $k - 1$ keys
- Logarithmic time $O(\log k)$ to delete the root (or maximum) key:
  - Remove the root key
  - Delete the leaf position k and move its key into the root
  - Bubble (percolate) the root key down by swapping with the largest child if the latter one is greater

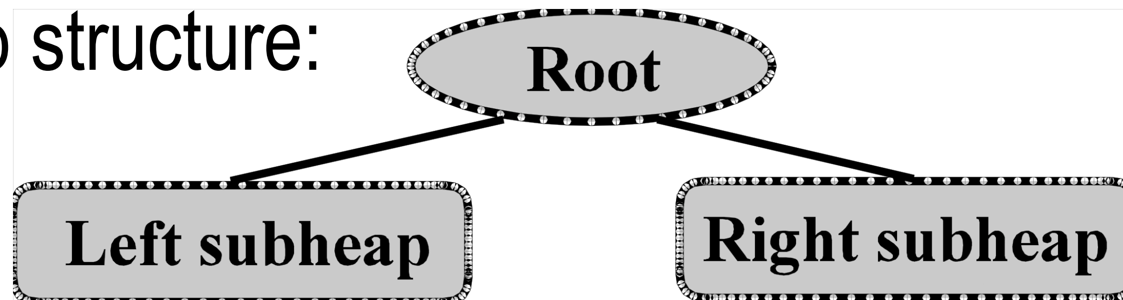


Binary Heap: An Example of Deleting the Maximum Key

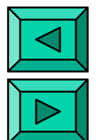


Linear Time Heap Construction

- n insertions take $O(n \log n)$ time.
- alternative $O(n)$ procedure uses a recursively defined heap structure:



  – form recursively the left and right subheaps
  – percolate the root down to establish the heap order everywhere



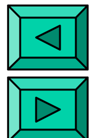
Heapifying Recursion

70 Root
65 50
20 2 91 25
31 15 8

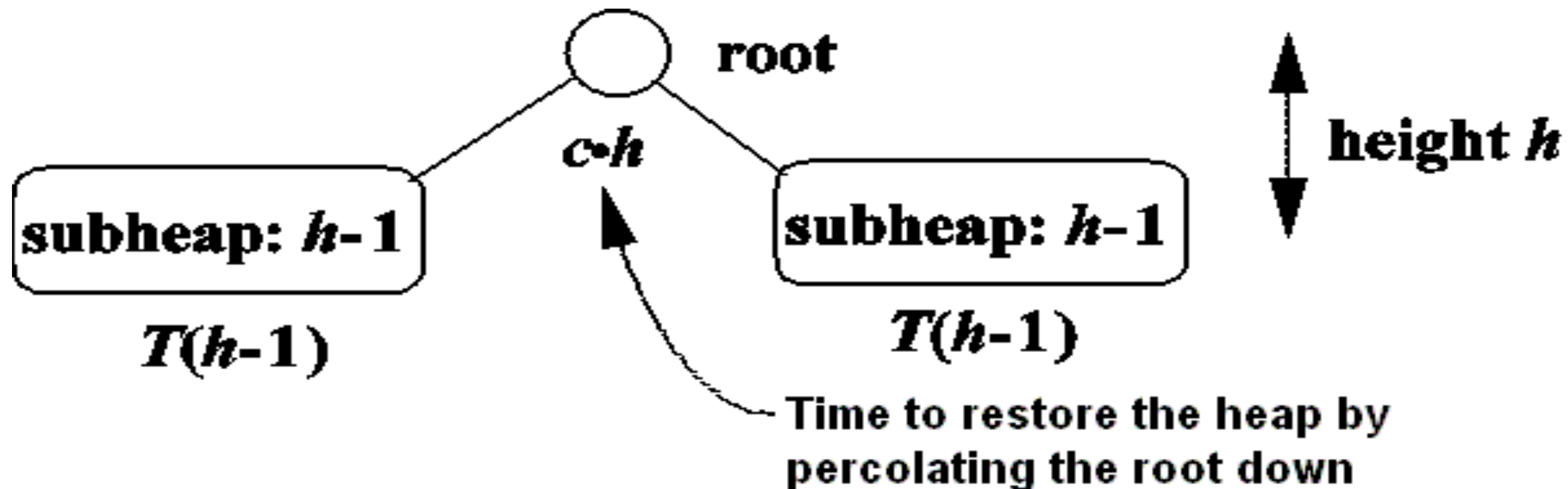
Leaves
recursive scan

h=1
h=2
h=3
h=4

91
s4
65 70
s3
31 8 50 25
s1 s2
20 15 2

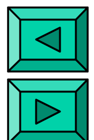


Time to Build a Heap



$$T(h) = 2\,T(h-1) + c \cdot h; \qquad T(0) = 0$$

$$\rightarrow \qquad T(h) = c \cdot \left(2^{h+1} - h - 1\right)$$

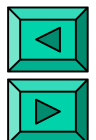


Linear Time Heap Construction: A Non-Recursive Procedure

- Nodes are percolated down in reverse level order
- When the node p is processed, its descendants will have been already processed.
- Leaves need not to be percolated down.
- Worst-case time $T(h)$ for building a heap of height h:

$$T(h) = 2T(h-1) + ch \rightarrow T(h) = O(2^h)$$

  - Form two subheaps of height $h-1$
  - Percolate the root down a path of length at most h



Time to build a heap

$$T(h) = 2\,T(h-1) + c \cdot h$$

$$2\,T(h-1) = 2^2\,T(h-2) + 2c \cdot (h-1)$$

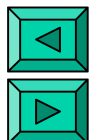
$$\cdots \quad \cdots \quad \cdots$$

$$2^{h-2}\,T(2) = 2^{h-1}\,T(1) + 2^{h-2} c \cdot 2$$

$$2^{h-1}\,T(1) = 2^{h}\,T(0) + 2^{h-1} c \cdot 1 = 2^{h-1} c \cdot 1$$

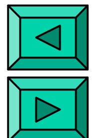
$$T(h) = c \cdot \left(1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + \ldots + (h-2) \cdot 2^2 + (h-1) \cdot 2^1 + h \cdot 2^0\right)$$

$$= c \cdot \left(2^{h+1} - h - 1\right)$$



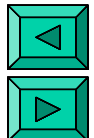
Steps of HeapSort

$p/_i$	$1/_0$	$2/_1$	$3/_2$	$4/_3$	$5/_4$	$6/_5$	$7/_6$	$8/_7$	$9/_8$	$10/_9$
a	70	65	50	20	2	91	25	31	15	8
HEA					8					2
PI				31				20		
F			91			50				
Y	91		70							
h	**91**	**65**	**70**	**31**	**8**	**50**	**25**	**20**	**15**	**2**



Steps of HeapSort

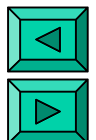
a_1	2	65	70	31	8	50	25	20	15	**91**
Restore the heap (R.h.)	70		2							
			50			2				
H_9	70	65	50	31	8	2	25	20	15	
a_2	15	65	50	31	8	2	25	20	**70**	**91**
R.h.	65	15								
		31		15						
				20				15		
h_8	65	31	50	20	8	2	25	15		



The University of Auckland

Steps of HeapSort

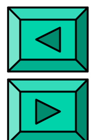
a_3	15	31	50	20	8	2	25	**65**	**70**	**91**
R.h.	50		15							
			25				15			
h_7	50	31	25	20	8	2	15			
a_4	15	31	25	20	8	2	**50**	**65**	**70**	**91**
R.h.	31	15								
		20		15						
h_6	31	20	25	15	8	2				



The University of Auckland

Steps of HeapSort

a_5	2	20	25	15	8	**31**	**50**	**65**	**70**	**91**
R. h.	25		2							
h_5	25	20	2	15	8					
a_6	8	20	2	15	**25**	**31**	**50**	**65**	**70**	**91**
R. h.	20	8								
		15		8						
h_4	20	15	2	8						

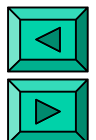


The University of Auckland

Steps of HeapSort

a_7	8	15	2	20	25	31	50	65	70	91
R. h.	15	8								
h_3	15	8	2							
a_8	2	8	15	20	25	31	50	65	70	91
R. h.	8	2								
h_2	8	2								
a_9	2	8	15	20	25	31	50	65	70	91

sorted array



BeadSort

- Parallel algorithm designed by Arulanandham, Calude and Dinneen (2002).
- Represent positive integers by a set of beads, like those used in an abacus. Beads are attached to vertical rods and appear to be suspended in the air just before sliding down (a number is read horizontally, as a row). After their falls, the rows of numbers have been rearranged such as the smaller numbers appears on top of greater numbers.
- Time: $O(\sqrt{n})$
- **www.cs.auckland.ac.nz/~jaru003/BeadSort.ppt**