Decrease & Conquer

Decrease-and-Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

Can be implemented either

- top-down (inductive/recursive approach)
- bottom-up (incremental/iterative approach)

Types of Decrease-and-Conquer

Decrease by a constant (usually by 1):

- insertion sort, shellsort

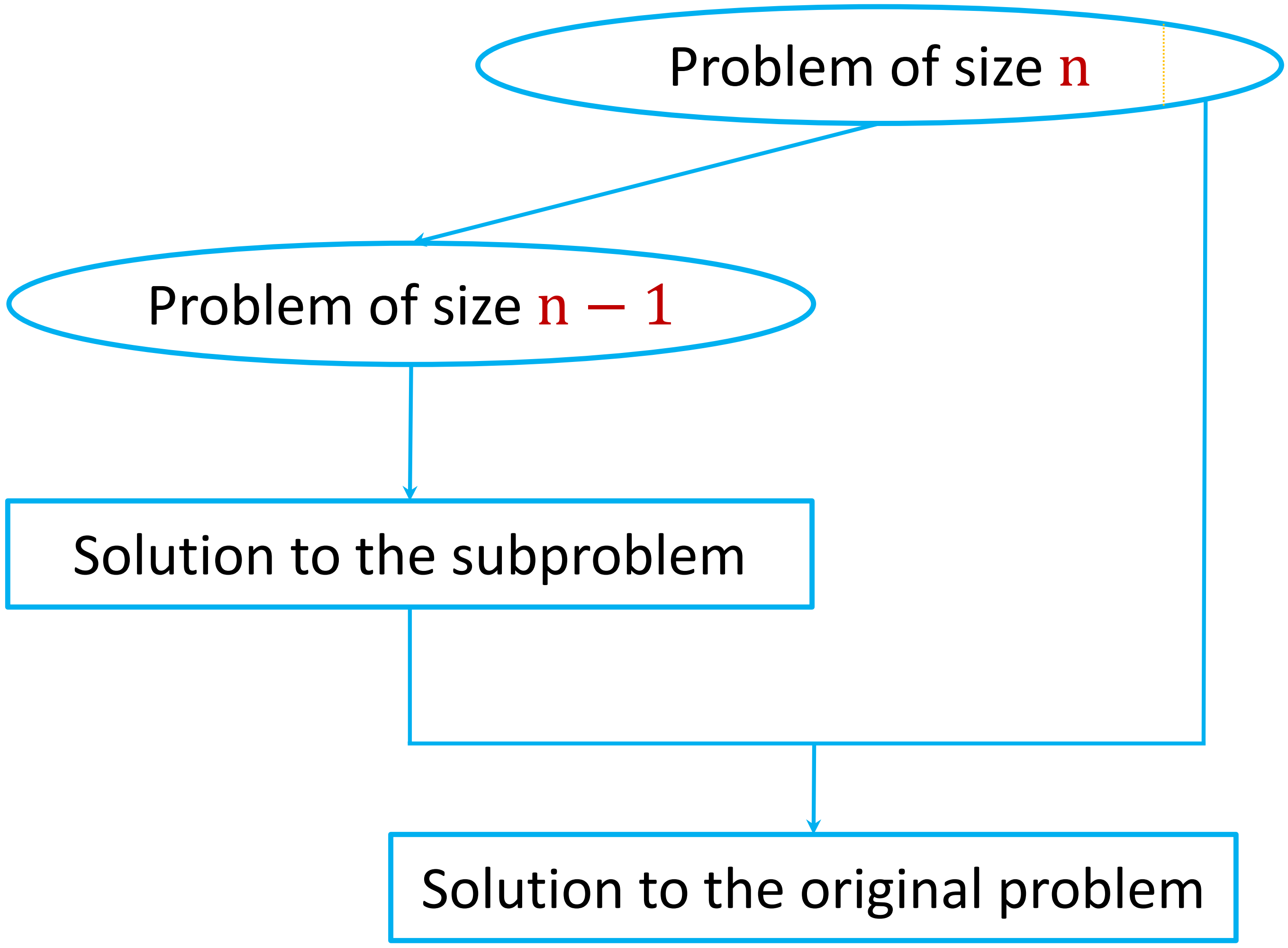
Decrease by a constant factor (usually by half)

- binary search
- Russian peasant multiplication

Variable-size decrease

- Euclid’s algorithm
- Interpolation search

Decrease-by-a-Constant



Types of Decrease-and-Conquer

Example: $a^n, a \neq 0, n \geq 0$

$$a^n = a^{n-1} \cdot a$$

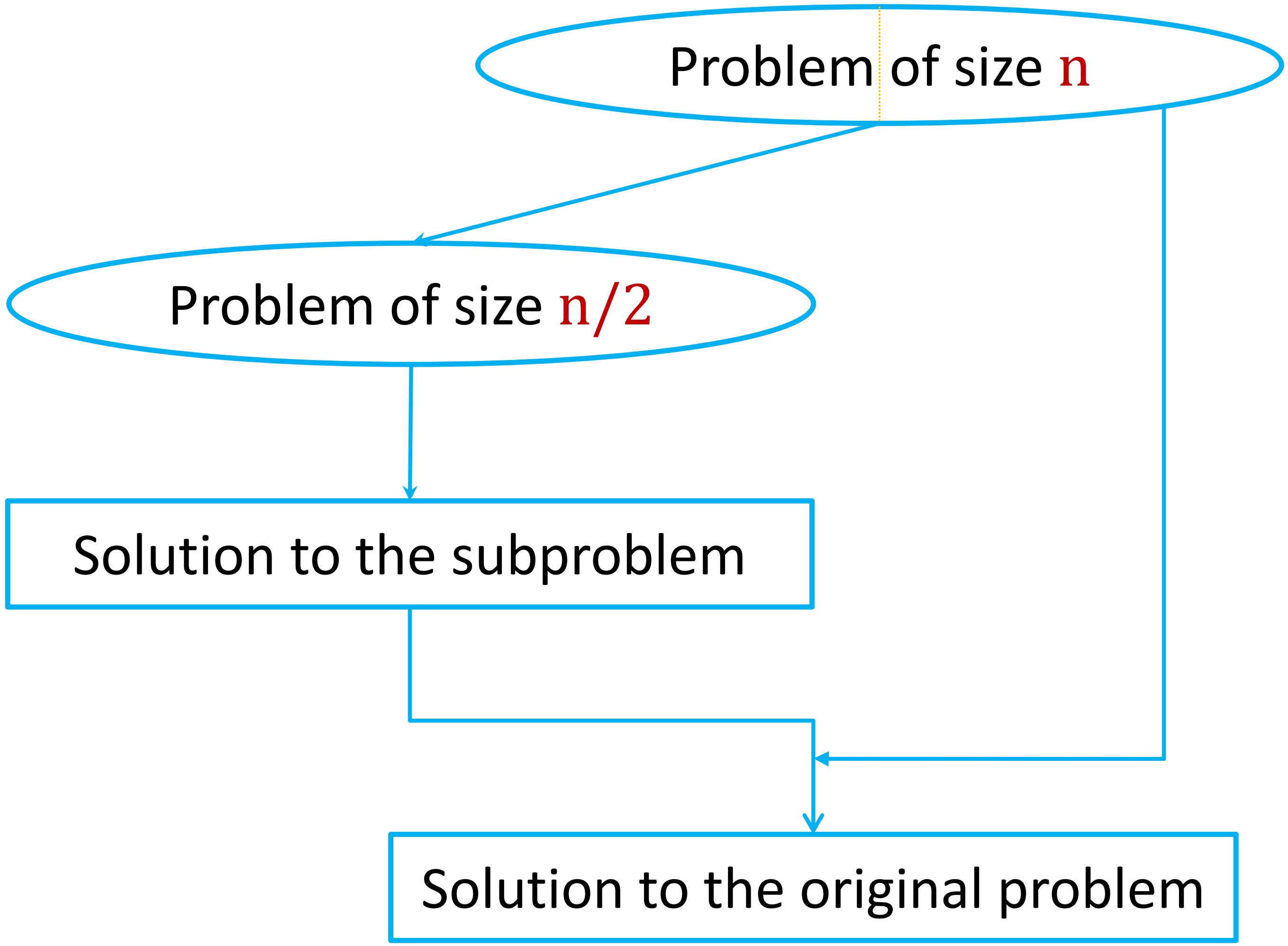
- Top-down:

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0 \\ 1, & \text{if } n = 0 \end{cases}$$

- Bottom-up:

$$f(1) = a$$
$$f(2) = f(1) \cdot a$$
$$f(3) = f(2) \cdot a$$
$$\ldots$$

Decrease-by-a-Constant Factor



Decrease-by-a-Constant Factor

Example: $a^n,\ a \neq 0,\ n \geq 0$

$$a^n = \left(a^{n/2}\right)^2$$

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0 \end{cases}$$

Algorithm: top-down (recursively)

Efficiency: $\Theta(\log n)$

The Variable Size Decrease

- The size-reduction pattern varies from one iteration of algorithm to another.

Example: Euclid's algorithm for computing the $\gcd(m, n)$

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

- **Time efficiency?**

The Variable Size Decrease

- The size-reduction pattern varies from one iteration of algorithm to another.

Example: Euclid's algorithm for computing the $\gcd(m, n)$

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

- **Time efficiency:** $T(n)$, the size, measured by the second number, **decreases at least by half** after two consecutive iterations. Hence, $T(n) = \Theta(\log n)$

Insertion Sort

To sort array $A[0..n-1]$,

- **sort** $A[0..n-2]$ **recursively** and
- then **insert** $A[n-1]$ **in its proper place among the sorted** $A[0..n-2]$:

  scan the sorted subarray **from right to left** until the **first element smaller than or <u>equal to</u>** $A[n-1]$ is encountered to **insert it after that element**.

- Usually implemented bottom up (nonrecursively)

Insertion Sort

Algorithm InsertionSort($A[0..n-1]$)

// Input: $A[0..n-1]$

//Output: $A[0..n-1]$ sorted in nondecreasing order

for $i \leftarrow 0$ to $n-1$ **do**

  $v \leftarrow A[i]$

  $j \leftarrow i-1$

  while $j > -1$ **and** $A[j] > v$ **do**

    $A[j+1] \leftarrow A[j]$

    $j \leftarrow j-1$

  $A[j+1] \leftarrow v$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

0	1	2	3	4	5	6
6	**4**	**1**	**7**	**3**	**2**	**5**

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

j	i						
-1	0	1	2	3	4	5	6
	6	**4**	**1**	**7**	**3**	**2**	**5**

$$v = A[0] = 6$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j	i					
-1	0	1	2	3	4	5	6
	6	**4**	**1**	**7**	**3**	**2**	**5**

$v = A[1] = 4$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

-1	0	1	2	3	4	5	6
	j	i					
	6	**6**	**1**	**7**	**3**	**2**	**5**

$$v = A[1] = 4$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

j		i					
-1	0	1	2	3	4	5	6
	6	**6**	**1**	**7**	**3**	**2**	**5**

$$v = A[1] = 4$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

j		i					
-1	0	1	2	3	4	5	6
	4	**6**	**1**	**7**	**3**	**2**	**5**

$$v = A[1] = 4$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

		j	i				
-1	0	1	2	3	4	5	6
	4	**6**	**1**	**7**	**3**	**2**	**5**

$v = A[2] = 1$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j	i					
-1	0	1	2	3	4	5	6
	4	**6**	**6**	**7**	**3**	**2**	**5**

$v = A[2] = 1$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

j (at index 0), i (at index 2)

-1	0	1	2	3	4	5	6
	4	**6**	**6**	**7**	**3**	**2**	**5**

$v = A[2] = 1$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j		i				
-1	0	1	2	3	4	5	6
	4	**4**	**6**	**7**	**3**	**2**	**5**

$v = A[2] = 1$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

j			i				
-1	0	1	2	3	4	5	6
	4	**4**	**6**	**7**	**3**	**2**	**5**

$$v = A[2] = 1$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

j			i				
-1	0	1	2	3	4	5	6
	1	**4**	**6**	**7**	**3**	**2**	**5**

$$v = A[2] = 1$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

			j	i			
-1	0	1	2	3	4	5	6
	1	**4**	**6**	**7**	**3**	**2**	**5**

$$v = A[3] = 7$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

				j	i		
-1	0	1	2	3	4	5	6
	1	**4**	**6**	**7**	**3**	**2**	**5**

$$v = A[4] = 3$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

				j	i		
-1	0	1	2	3	4	5	6
	1	**4**	**6**	**7**	**7**	**2**	**5**

$$v = A[4] = 3$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

			j		i		
-1	0	1	2	3	4	5	6
	1	**4**	**6**	**7**	**7**	**2**	**5**

$$v = A[4] = 3$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

			j		i		
-1	0	1	2	3	4	5	6
	1	**4**	**6**	**6**	**7**	**2**	**5**

$$v = A[4] = 3$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

		j			i		
-1	0	1	2	3	4	5	6
	1	**4**	**6**	**6**	**7**	**2**	**5**

$$v = A[4] = 3$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

		j			i		
-1	0	1	2	3	4	5	6
	1	**4**	**4**	**6**	**7**	**2**	**5**

$$v = A[4] = 3$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j				i		
-1	0	1	2	3	4	5	6
	1	**4**	**4**	**6**	**7**	**2**	**5**

$$v = A[4] = 3$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j				i		
-1	0	1	2	3	4	5	6
	1	**3**	**4**	**6**	**7**	**2**	**5**

$$v = A[4] = 3$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

					j	i	
-1	0	1	2	3	4	5	6
	1	**3**	**4**	**6**	**7**	**2**	**5**

$$v = A[5] = 2$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

					j	i	
-1	0	1	2	3	4	5	6
	1	**3**	**4**	**6**	**7**	**7**	**5**

$$v = A[5] = 2$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j				i	
0	1	2	3	4	5	6
1	**3**	**4**	**6**	**7**	**7**	**5**

-1

$$v = A[5] = 2$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

				j		i	
-1	0	1	2	3	4	5	6
	1	**3**	**4**	**6**	**6**	**7**	**5**

$$v = A[5] = 2$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j				i	

-1	0	1	2	3	4	5	6
	1	**3**	**4**	**6**	**6**	**7**	**5**

$v = A[5] = 2$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j				i	

-1	0	1	2	3	4	5	6
	1	**3**	**4**	**4**	**6**	**7**	**5**

$$v = A[5] = 2$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j				i	

-1	0	1	2	3	4	5	6
	1	**3**	**4**	**4**	**6**	**7**	**5**

$\mathrm{v} = A[5] = 2$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j				i	
0	1	2	3	4	5	6
1	**3**	**3**	**4**	**6**	**7**	**5**

-1

$$v = A[5] = 2$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j					i	
-1	0	1	2	3	4	5	6
	1	**3**	**3**	**4**	**6**	**7**	**5**

$$v = A[5] = 2$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

	j					i	
-1	0	1	2	3	4	5	6
	1	**2**	**3**	**4**	**6**	**7**	**5**

$$v = A[5] = 2$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

						j	i
-1	0	1	2	3	4	5	6
	1	**2**	**3**	**4**	**6**	**7**	**5**

$$v = A[6] = 5$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

						j	i
-1	0	1	2	3	4	5	6
	1	**2**	**3**	**4**	**6**	**7**	**7**

$$v = A[6] = 5$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

					j		i
-1	0	1	2	3	4	5	6
	1	**2**	**3**	**4**	**6**	**7**	**7**

$v = A[6] = 5$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

					j		i
-1	0	1	2	3	4	5	6
	1	**2**	**3**	**4**	**6**	**6**	**7**

$$v = A[6] = 5$$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

				j			i
-1	0	1	2	3	4	5	6
	1	**2**	**3**	**4**	**6**	**6**	**7**

$\mathrm{v} = \mathrm{A}[6] = 5$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

				j			i
-1	0	1	2	3	4	5	6
	1	**2**	**3**	**4**	**5**	**6**	**7**

$v = A[6] = 5$

Insertion Sort

Example: Sort 6, 4, 1, 7, 2, 5, 3

0	1	2	3	4	5	6
1	**2**	**3**	**4**	**5**	**6**	**7**

Insertion Sort

Algorithm InsertionSort$(A[0..n-1])$

// Input: $A[0..n-1]$

//Output: $A[0..n-1]$ sorted in nondecreasing order

for $i \leftarrow 0$ to $n-1$ **do**

$v \leftarrow A[i]$

$j \leftarrow i-1$

while $j \geq 0$ **and** $A[j] > v$ **do**

$A[j+1] \leftarrow A[j]$

$j \leftarrow j-1$

$A[j+1] \leftarrow v$

Analysis of Insertion Sort

- The basic operation is the key comparison

$$A[j] > v$$

- The key comparisons depend on the nature of the input

- Worst-case: The comparison $A[j] > v$ is executed for every $j = i - 1, \ldots, 0$ in each iteration of the outer loop:

$$A[0] > A[1] > \cdots > A[n-1]$$

$$C_w(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \cdots = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Analysis of Insertion Sort

- The basic operation is the key comparison

$$A[j] > v$$

- The key comparisons depend on the nature of the input

- Best-case: The comparison $A[j] > v$ is executed only once in each iteration of the outer loop:

$$A[0] \leq A[1] \leq \cdots \leq A[n-1]$$

$$C_b(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Analysis of Insertion Sort

- **Time efficiency:**
  - $C_w(n) = \Theta(n^2)$
  - $C_a(n) = \Theta(n^2)$
  - $C_b(n) = \Theta(n)$
- **Space efficiency:** in-place
- **Stability:** yes
- **Best elementary sorting algorithm overall**
- **Improvement:** binary insertion sort, **shellsort**

Shellsort

- Move entries more than one position at a time by h-sorting the array
- An h-sorted array is h **interleaved sorted subsequences**
- $h = 4$

L	**E**	**E**	**A**	**M**	**H**	**L**	**E**	**P**	**S**	**O**	**L**	**T**	**S**	**X**	**R**

Shellsort

- Move entries more than one position at a time by h-sorting the array
- An h-sorted array is **h interleaved sorted subsequences**
- $h = 4$

L	**E**	**E**	**A**	**M**	**H**	**L**	**E**	**P**	**S**	**O**	**L**	**T**	**S**	**X**	**R**
L				**M**				**P**				**T**			

Shellsort

- Move entries more than one position at a time by h-sorting the array
- An h-sorted array is **h interleaved sorted subsequences**
- $h = 4$

L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
	E				H				S				S		

Shellsort

- Move entries more than one position at a time by h-sorting the array
- An h-sorted array is **h interleaved sorted subsequences**
- $h = 4$

L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
		E				L				O				X	

Shellsort

- Move entries more than one position at a time by h-sorting the array
- An h-sorted array is **h interleaved sorted subsequences**
- $h = 4$

L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
			A				E				L				R

Shellsort

- Move entries more than one position at a time by h-sorting the array
- An h-sorted array is **h interleaved sorted subsequences**
- $h = 4$

L E E A M H L E P S O L T S X R

Shellsort

- **Shellsort** [Shell 1959]: **h-sort array for decreasing sequence of values of h.**

Input:

S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E

Shellsort

- **Shellsort [Shell 1959]: h-sort array for decreasing sequence of values of h.**

Input:

S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E

13-sort:

P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E

Shellsort

- **Shellsort** [Shell 1959]: h-sort array for decreasing sequence of values of h.

Input:

S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E

13-sort:

P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E

4-sort:

L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R

Shellsort

- **Shellsort** [Shell 1959]: h-sort array for decreasing sequence of values of h.

Input:

S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E

13-sort:

P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E

4-sort:

L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R

1-sort:

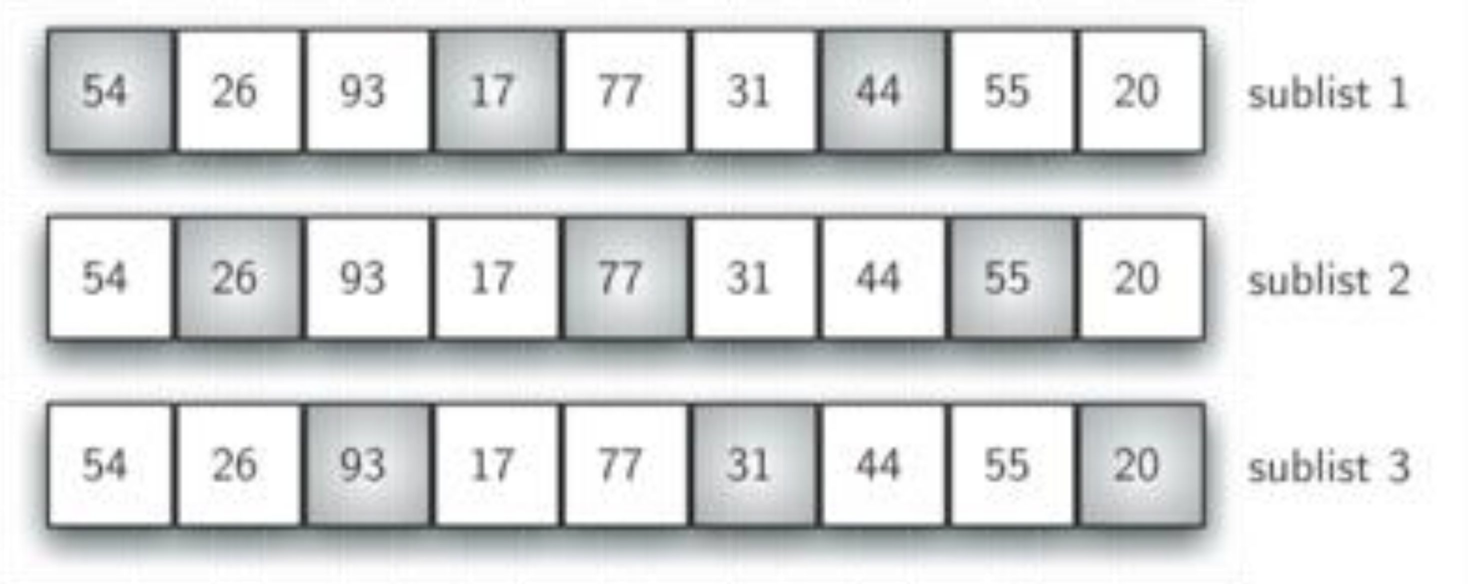
A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Shellsort

- The **shell sort** (“**diminishing increment sort**”), improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort.

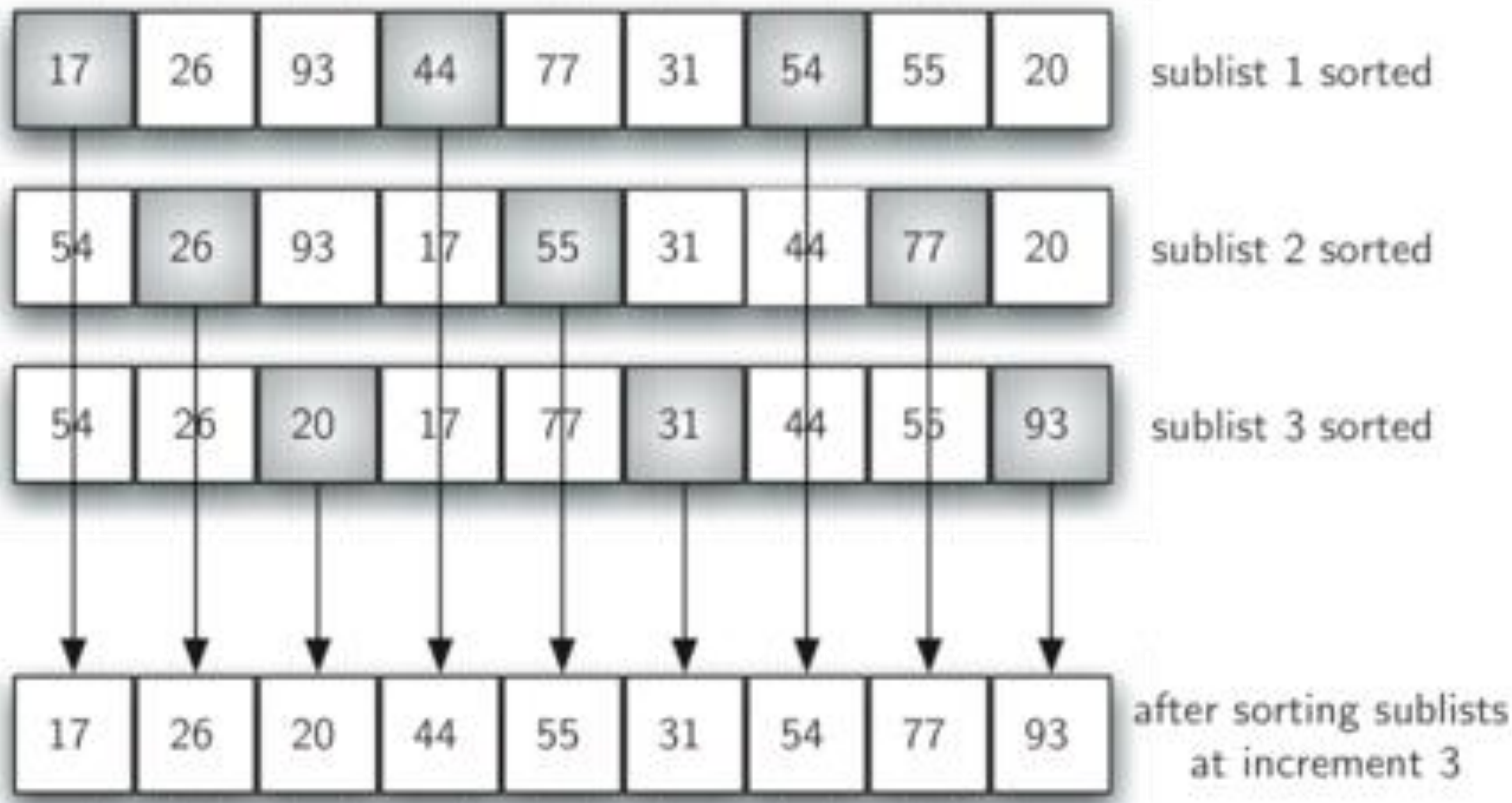
- Instead of breaking the list into sublists of contiguous items, the shell sort uses an **increment** i (**gap**), to create a sublist by choosing all items that are i items apart.

Shellsort

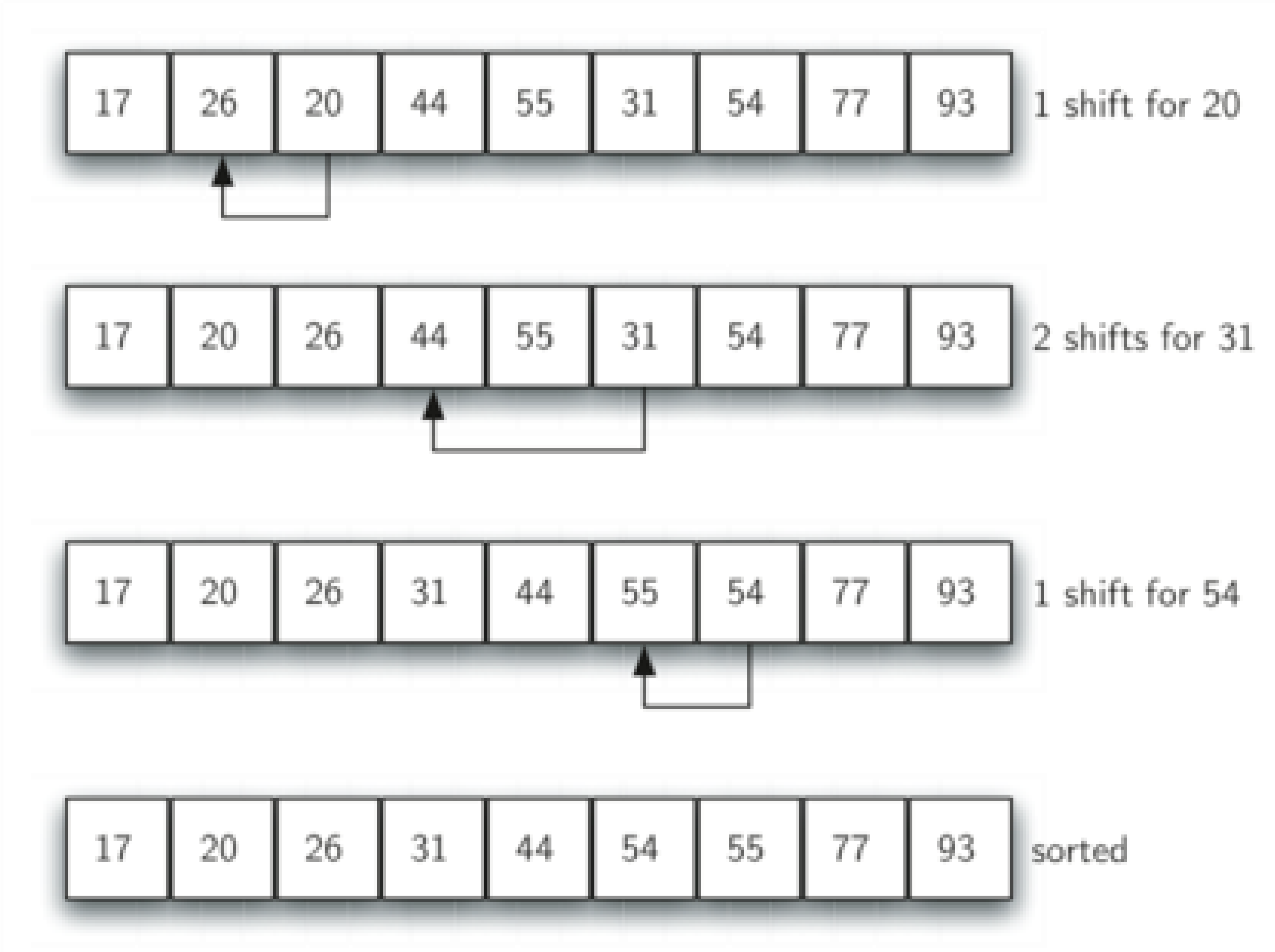


54
26
93
17
77
31
44
55
20
sublist 1
54
26
93
17
77
31
44
55
20
sublist 2
54
26
93
17
77
31
44
55
20
sublist 3

Shellsort

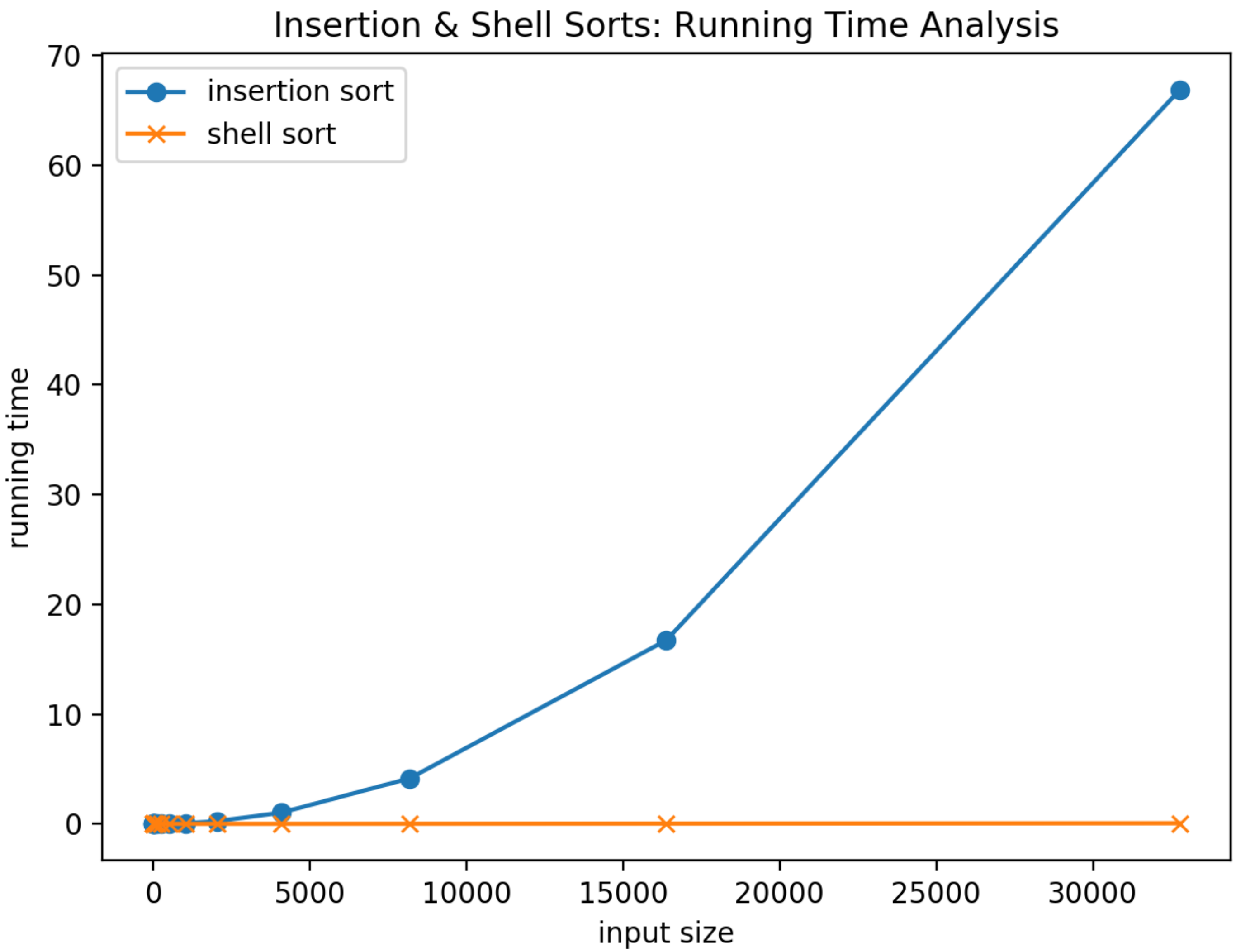

17 26 93 44 77 31 54 55 20
sublist 1 sorted
54 26 93 17 55 31 44 77 20
sublist 2 sorted
54 26 20 17 77 31 44 55 93
sublist 3 sorted
17 26 20 44 55 31 54 77 93
after sorting sublists at increment 3

Shellsort



17 26 20 44 55 31 54 77 93
1 shift for 20
17 20 26 44 55 31 54 77 93
2 shifts for 31
17 20 26 31 44 55 54 77 93
1 shift for 54
17 20 26 31 44 54 55 77 93
sorted

Shellsort


Insertion & Shell Sorts: Running Time Analysis
insertion sort
shell sort
running time
input size
70
60
50
40
30
20
10
0
0
5000
10000
15000
20000
25000
30000

Summary

ALGORITHM	BEST	AVERAGE	WORST
Selection sort	n^2	n^2	n^2
Bubble sort	n^2	n^2	n^2
Insertion sort	n	n^2	n^2
Shellsort	n	?	$n^{1.5}$

Decrease-by-a-Constant-Factor

Binary search is an **efficient algorithm** for searching in a sorted array:

- it compares a search key K with the array's middle element $\mathrm{A[m]}$
- if they match, the algorithm stops
- otherwise, the same operation is repeated recursively for the first half of the array if $\mathrm{K} < \mathrm{A[m]}$, or for the second half if $\mathrm{K} > \mathrm{A[m]}$

Binary Search

$$\underbrace{\mathrm{A}[0] \quad \dots \quad \mathrm{A}[\mathrm{m}-1]}_{\substack{\text{Search here if} \\ \mathrm{K} < \mathrm{A}[\mathrm{m}]}} \quad \mathrm{A}[\mathrm{m}] \quad \underbrace{\mathrm{A}[\mathrm{m}+1] \quad \dots \quad \mathrm{A}[\mathrm{n}-1]}_{\substack{\text{Search here if} \\ \mathrm{K} > \mathrm{A}[\mathrm{m}]}}$$

$$\updownarrow$$

$$\mathrm{K}$$

Binary Search

Algorithm BinarySearch$(A[0..n-1], K)$

// Input: Array $A[0..n]$ and search key K

//Output: The index m, where $A[m] = K$ or -1

$l \leftarrow 0;\;\; r \leftarrow n-1$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l+r)/2 \rfloor$

if $K = A[m]$ **return** m

else if $K < A[m]$ $\quad r \leftarrow m-1$

else $l \leftarrow m+1$

return -1

Binary Search

$K = 70$

l r

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98

Binary Search

$$K = 70$$

l						m						r
0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98

$$m = \left\lfloor \frac{(0 + 12)}{2} \right\rfloor = 6$$

Binary Search

$K = 70$

							l					r
0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98

Binary Search

$$K = 70$$

							l		m			r
0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98

$$m = \left\lfloor \frac{(7+12)}{2} \right\rfloor = 9$$

Binary Search

$K = 70$

							l	r				
0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98

Binary Search

$$K = 70$$

							l,m	r				
0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98

$$m = \left\lfloor \frac{(7+8)}{2} \right\rfloor = 7$$

Binary Search

$K = 70$

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	**70**	74	81	85	93	98

Analysis of Binary Search

- count **the number of times the search key is compared with an element of the array** (3-way comparison: after one comparison K with $\mathrm{A[m]}$ the algorithm can determine if $\mathrm{K < A[m]}$, $\mathrm{K = A[m]}$ or $\mathrm{K > A[m]}$)

- The number of comparisons depends not only n but also on the **specifics of a particular instance** of the problem

- **Worst-case**: inputs include all array not containing a given search key & some successful searches.

Analysis of Binary Search

After one comparison the algorithm gets an array half size

$$C_w(n) = C_w(\lfloor n/2 \rfloor) + 1, n \geq 1$$

$$C_w(1) = 1$$

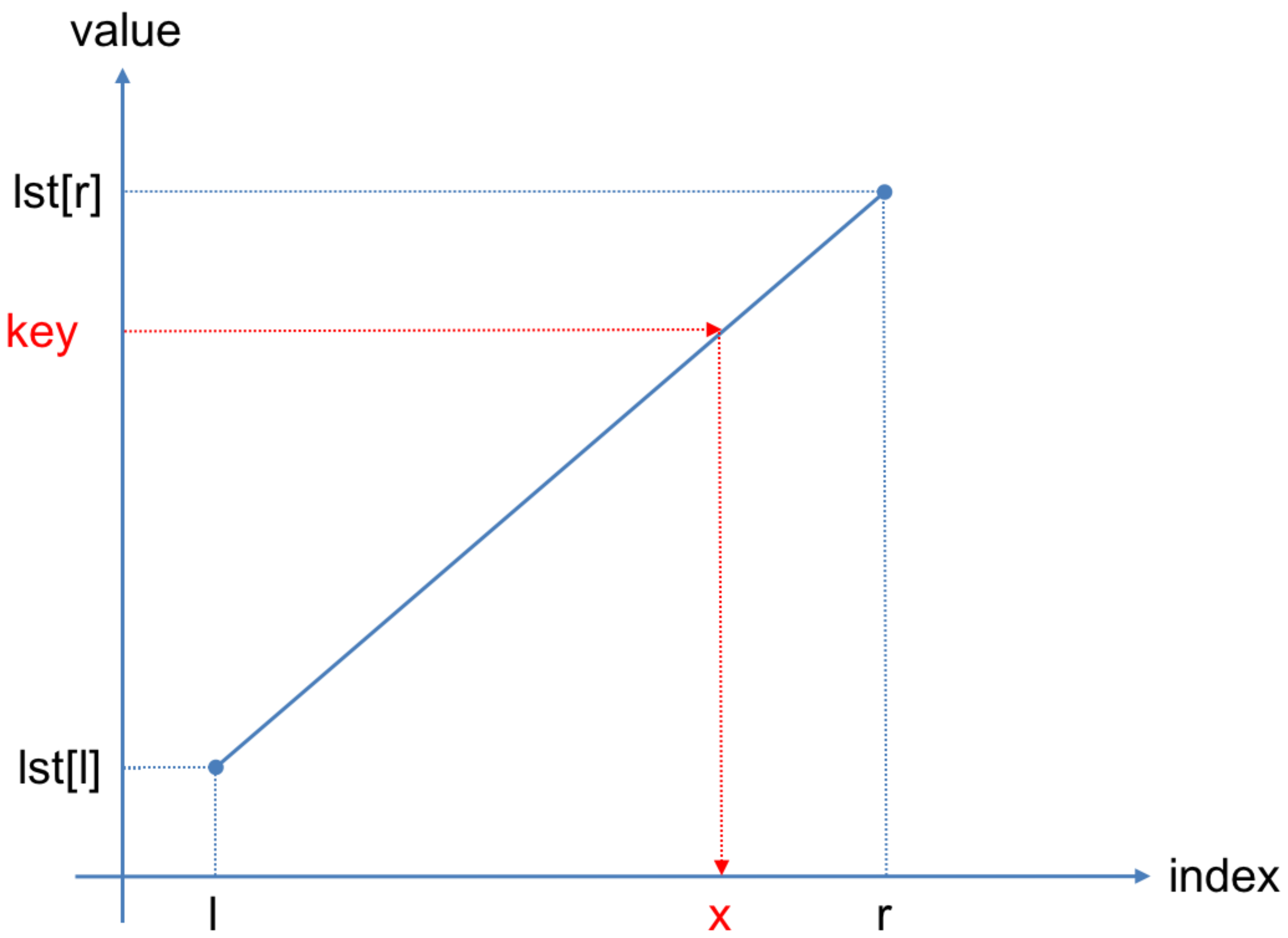
- $C_w = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil \in \Theta(\log n)$
- $C_a \approx \log_2 n \in \Theta(\log n)$

Interpolation Search

- **Interpolation search algorithm** searches a sorted list similar to binary search but estimates location of the search item in $\mathrm{lst}[\mathrm{l}..\mathrm{r}]$ by using its value key.
- The values of the list elements are assumed to grow linearly from $\mathrm{lst}[\mathrm{l}]$ to $\mathrm{lst}[\mathrm{r}]$.
- The location of key key is estimated as the x-coordinate of the point on the straight line through $(\mathrm{l}, \mathrm{lst}[\mathrm{l}])$ and $(\mathrm{r}, \mathrm{lst}[\mathrm{r}])$ whose y-coordinate is key:

$$\mathrm{x} = \mathrm{l} + \frac{(\mathrm{key} - \mathrm{lst}[\mathrm{l}]) \times (\mathrm{r} - \mathrm{l})}{\mathrm{lst}[\mathrm{r}] - \mathrm{lst}[\mathrm{l}]}$$

Interpolation Search



Interpolation Search: Analysis

Efficiency:

- **average case:**

$$T_{avg}(n) < \log\log n + 1$$

- **worst case:**

$$T_{worst}(n) = n$$

- Preferable to binary search only for **very large lists** and/or **expensive comparisons**

Russian Peasant Multiplication

Problem: Compute the product of two positive integers n and m

- Can be solved by a decrease-by-half algorithm:

$$n \cdot m = \begin{cases} \frac{n}{2} \cdot (2 \cdot m) & \text{if } n > 1 \text{ is even} \\ \frac{(n-1)}{2} \cdot (2 \cdot m) + m & \text{if } n > 1 \text{ is odd} \\ m & \text{if } n = 1 \end{cases}$$

Russian Peasant Multiplication

Compute $20 \cdot 26$

n	m
20	26

Russian Peasant Multiplication

Compute $20 \cdot 26$

n	m
20	26
10	52

Russian Peasant Multiplication

Compute $20 \cdot 26$

n	m
20	26
10	52
5	104

Russian Peasant Multiplication

Compute $20 \cdot 26$

n	m
20	26
10	52
5	104

Russian Peasant Multiplication

Compute $20 \cdot 26$

n	m	
20	26	
10	52	
5	104	104
2	208	

Russian Peasant Multiplication

Compute $20 \cdot 26$

n	m	
20	26	
10	52	
5	104	104
2	208	
1	416	

Russian Peasant Multiplication

Compute $20 \cdot 26$

n	m	
20	26	
10	52	
5	104	104
2	208	
1	416	416

Russian Peasant Multiplication

Compute $20 \cdot 26$

n	m	
20	26	
10	52	
5	104	104
2	208	+
1	416	416
		520

Fake-Coin Puzzle

- There are n **identically looking coins** one of which is **fake**. There is a balance scale but there are no weights; the scale can tell whether two sets of coins weigh the same and, if not, which of the two sets is heavier (but not by how much).

- **Design an efficient algorithm for detecting the fake coin.** Assume that the fake coin is known to be lighter than the genuine ones.