Chapter 7
Sorting

Introduction

- sorting
  - fundamental task in data management
  - well-studied problem in computer science
- basic problem
  - given an <u>array</u> of items where each item contains a key, rearrange the items so that the keys appear in ascending order
  - the key may be only <u>part</u> of the item begin sorted
    - e.g., the item could be an entire block of information about a student, while the search key might be only the student's name

Introduction

- we will assume
  - the array to sort contains only integers
  - defined < and > operators (comparison-based sorting)
  - all N array positions contain data to be sorted
  - the entire sort can be performed in main memory
    - number of elements is relatively small: < a few million
- given a list of n items, $a_0, a_1, \ldots, a_{n-1}$, we will use the notation $a_i \leq a_j$ to indicate that the search key for a a_i does not follow that of a_j
- sorts that cannot be performed in main memory
  - may be performed on disk or tape
  - known as external sorting

Sorting Algorithms

- $O(N^2)$ sorts
  - insertion sort
  - selection sort
  - bubble sort
  - why $O(N^2)$?
- Shellsort: <u>subquadratic</u>
- $O(N \lg N)$ sorts
  - heapsort
  - mergesort
  - quicksort
  - a lower bound on sorting by <u>pairwise</u> comparison
- $O(N)$ sorting algorithms: count sort
- string sorts

Sorting

- given N keys to sort, there are $N!$ possible permutations of the keys!
  - e.g, given $N = 3$ and the keys a, b, c, there are $3! = 3 \cdot 2 \cdot 1 \; = \; 6$ possible permutations:

$$abc \;\; acb \;\; bac \;\; bca \;\; cab \;\; cba$$

  - brute force enumeration of permutations is not computationally feasible once $N > 10$
    - $13! = 6.2270 \times 10^9$
    - $20! = 2.4329 \times 10^{18}$

Sorting

- cost model for sorting
  - the basic operations we will count are <u>comparisons</u> and <u>swaps</u>
  - if there are array accesses that are not associated with comparisons or swaps, we need to count them, too
- programming notes
  - if the objects being sorted are large, we should swap <u>pointers</u> to the objects, rather than the objects themselves
  - polymorphism: general-purpose sorting algorithms vs templated algorithms

Insertion Sort

- insertion sort
  - simple
- $N-1$ passes
  - in pass p, $p = 1, \ldots, N-1$, we move a_p to its correct location among $a_0, \ldots, a_p$
  - for passes $p = 1$ to $N-1$, insertion sort ensures that the elements in positions 0 to p are in sorted order
  - at the end of pass p, the elements in positions 0 to p are in sorted order

Insertion Sort

- algorithm

```
for (p = 1; p < N; p++) {
    tmp = a_p
    for (j = p; (j > 0) && (tmp < a_{j-1}); j--) {
        swap a_j and a_{j-1}
    }
    a_j = tmp
}
```

Insertion Sort

– to avoid the full swap, we can use the following code:

```
for (p = 1; p < N; p++) {
    tmp = a_p
    for (j = p; (j > 0) && (tmp < a_{j-1}); j--) {
        a_j = a_{j-1}
    }
    a_j = tmp
}
```

Insertion Sort

– example

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Insertion Sort

- example

position	0	1	2	3	4	5
initial sequence	42	6	1	54	0	7
p = 1	6	42	1	54	0	7
p = 2	6	1	42	54	0	7
	1	6	42	54	0	7
p = 3	1	6	42	54	0	7
p = 4	1	6	42	0	54	7
	1	6	0	42	54	7
	1	0	6	42	54	7
	0	1	6	42	54	7
p = 5	0	1	6	42	7	54
	0	1	6	7	42	54

Insertion Sort

- analysis
  - best case: the keys are already sorted – $n - 1$ comparisons, no swaps
  - worst case: the keys are in reverse sorted order
  - expected (average) case: ?

Insertion Sort

- analysis
  - given $a_0, \ldots, a_{n-1}$ that we wish to sort in ascending order, an inversion is any pair that is out of order relative to one another: (a_i, a_j) for which $i < j$ but $a_i > a_j$
  - the list

$$42, 6, 9, 54, 0$$

  contains the following inversions:

$$(42, 6), (42, 9), (42, 0), (6, 0), (9, 0), (54, 0)$$

  - swapping an adjacent pair of elements that are out of order removes exactly one inversion
    - thus, any sort that operates by swapping adjacent terms requires as many swaps as there are inversions

Insertion Sort

- number of inversions
  - a list can have between 0 and $N(N-1)/2$ inversions, the latter of which occurs when $a_0 > a_1 > \cdots > a_{N-1}$
  - thus, counting inversions also says the worst-case behavior of insertion is quadratic
- what do inversions tell us about the expected behavior?
  - let P be the probability space of all permutations of N distinct elements with equal probability
    - Theorem: The expected number of inversions in a list taken from P is $N(N-1)/4$
    - thus, the expected complexity of insertion sort is quadratic

Insertion Sort

- proof
  - observe that any pair in a list that is an inversion is in the correct order in the <u>reverse</u> of that list

list	inversions	reverse lists	inversions
1, 2, 3, 4	0	4, 3, 2, 1	6
2, 1, 3, 4	1	4, 3, 1, 2	5
3, 2, 1, 4	3	4, 1, 2, 3	3

  - this means that if we look at a list and its reverse and count the total number of inversions, then the combined number of inversions is $N(N-1)/2$

Insertion Sort

- proof (cont.)
  - since there are $N!/2$ distinct pairs of lists and their reverses, there is a total of

$$\frac{N!}{2}\frac{N(N-1)}{2} = N!\frac{N(N-1)}{4}$$

inversions among the $N!$ possible lists of N distinct objects

  - this means that the expected number of inversions in any given list is $N(N-1)/4$

Insertion Sort

- in summary
  - for randomly ordered arrays of length N with distinct keys, insertion sort uses
    - $\sim N^2/4$ comparisons and $\sim N^2/4$ swaps on <u>average</u>
    - $\sim N^2/2$ comparisons and $\sim N^2/2$ swaps in the <u>worst</u> case
    - $N - 1$ comparisons and 0 swaps in the <u>best</u> case

Selection Sort

- selection sort
  - find the smallest item and exchange it with the first entry
  - find the next smallest item and exchange it with the second entry
  - find the next smallest item and exchange it with the third entry

  ...

Selection Sort

– algorithm

```
for (p = 0; p < N; p++) {
    m = p
    for (j = p+1; j < N; j++) {
        if (a_j < a_m) {
            m = j
        }
    }
    swap a_m and a_p
}
```

Selection Sort

- example

position	0	1	2	3	4
initial sequence	6	42	9	54	0
after p = 0	0	42	9	54	6
after p = 1	0	6	9	54	42
after p = 2	0	6	9	54	42
after p = 3	0	6	9	42	54

Selection Sort

- complexity
  - $N^2/2$ comparisons and N swaps to sort an array of length N
  - the amount of work is <u>independent</u> of the input
  - selection sort is no faster on <u>sorted</u> input than on random input
  - selection sort involves a smaller number of <u>swaps</u> than any of the other sorting algorithms we will consider

Selection Sort

- proof
  - there is one swap per iteration of the <u>outermost</u> loop, which is executed N times
  - there is one comparison made in each iteration of the <u>innermost</u> loop, which is executed

$$\sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1} 1 = \sum_{i=0}^{N-1}\big((N-1)-(i+1)+1\big) = \sum_{i=0}^{N-1}(N-i+1)$$

$$= N^2 - \frac{(N-1)N}{2} + N = \frac{N^2}{2} + \frac{3N}{2} \sim \frac{N^2}{2}$$

Bubble Sort

- bubble sort
  - read the items from left to right
  - if two adjacent items are out of order, swap them
  - repeat until sorted
  - a sweep with no swaps means we are done

Bubble Sort

– algorithm

```
not_done = true
while (not_done) {
    not_done = false
    for (i = 0 to N - 2) {
        if (a_i > a_{i+1}) {
            swap a_i and a_j
            not_done = true
        }
    }
}
```

Bubble Sort

– example

initial sequence	42	6	9	54	0
while-loop	6	42	9	54	0
	6	9	42	54	0
	6	9	42	0	54
while-loop	6	9	0	42	54
while-loop	6	0	9	42	54
while-loop	0	6	9	42	54

Bubble Sort

- complexity
  - if the data are already sorted, we make only one sweep through the list
  - otherwise, the complexity depends on the number of times we execute the while-loop
  - since bubble sort swaps adjacent items, it will have quadratic worst-case and expected-case complexity

Shellsort

- Shellsort
  - Donald L. Shell (1959), *A high-speed sorting procedure*, Communications of the ACM 2 (7): 3032.
  - swapping only adjacent items dooms us to quadratic worst-case behavior, so swap non-adjacent items!
  - Shellsort starts with an increment sequence

$$h_t > h_{t-1} > \cdots > h_2 > h_1 = 1$$

  - it uses insertion sort to sort
    - every h_t-th term starting at a_0, then $a_0, \ldots,$ then $a_{h_t - 1}$
    - every h_{t-1}-th term starting at a_0, then $a_0, \ldots,$ then $a_{h_{t-1} - 1}$
    - etc.
    - every term $(h_1 = 1)$ starting at a_0, after which the array is sorted

Shellsort

- Shellsort
  - suppose we use the increment sequence 15, 7, 5, 3, 1, and have finished the 15-sort and 7-sort
  - then we know that

$$a_0 \leq a_{15} \leq a_{30} \leq \cdots$$
$$a_0 \leq a_7 \leq a_{14} \leq \cdots$$

  - we also know that

$$a_{15} \leq a_{22} \leq a_{29} \leq a_{36} \leq \cdots$$

  - putting these together, we see that

$$a_0 \leq a_7 \leq a_{22} \leq a_{29} \leq a_{36}$$

Shellsort

- Shellsort
  - after we have performed the sort using increment h_k, the array is h_k-sorted: all elements that are h_k terms apart are in the correct order:

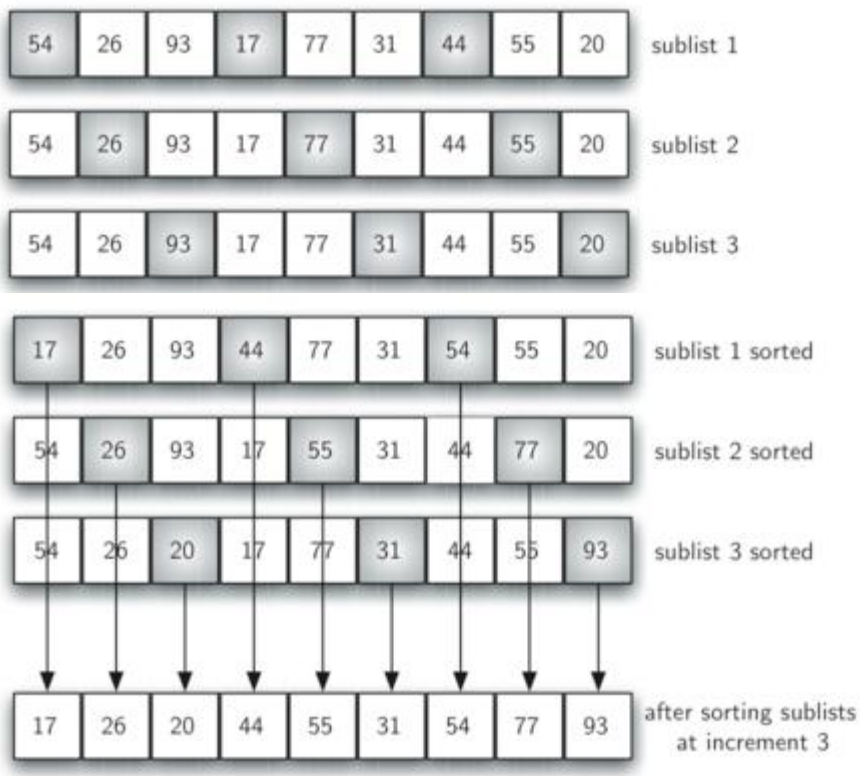
$$a_i \leq a_{i+h_k}$$

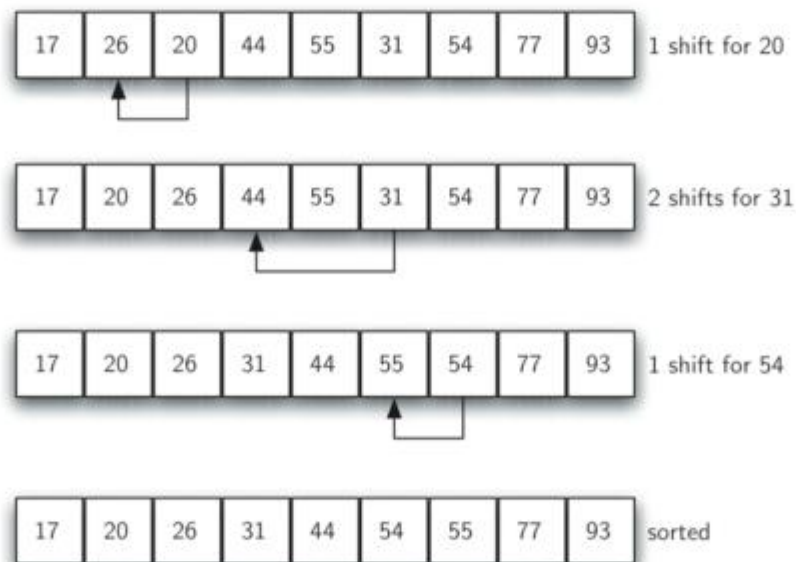
  - the key to Shellsort's efficiency is the following fact: an h_k-sorted array remains h_k-sorted after sorting with increment h_{k-1}

Shellsort

– examples

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96





http://interactivepython.org/KKOkZ/courselib/static/pythonds/SortSearch/TheShellSort.html

Shellsort

- complexity
  - a good increment sequence $h_t, h_{t-1}, \ldots, h_1 = 1$ has the property that for any element a_p, when it is time for the h_k-sort, there are only a few elements to the left of p that are larger than a_p
  - Shell's original increment sequence has N^2 worst-case behavior:

$$h_t = \left\lfloor \frac{N}{2} \right\rfloor, \; h_k = \left\lfloor \frac{h_{k+1}}{2} \right\rfloor$$

Shellsort

- complexity (cont.)
  - the sequences

$$2^k - 1 = 1, 3, 7, 15, 31, \ldots, \quad \text{(T. H. Hibbard, 1963)}$$

$$\frac{3^k - 1}{2} = 1, 4, 13, 40, 121 \text{ (V. R. Pratt, 1971)}$$

  yield $O\left(N^{3/2}\right)$ worst-case complexity

  - other sequences yield $O\left(N^{4/3}\right)$ worst-case complexity

Heapsort

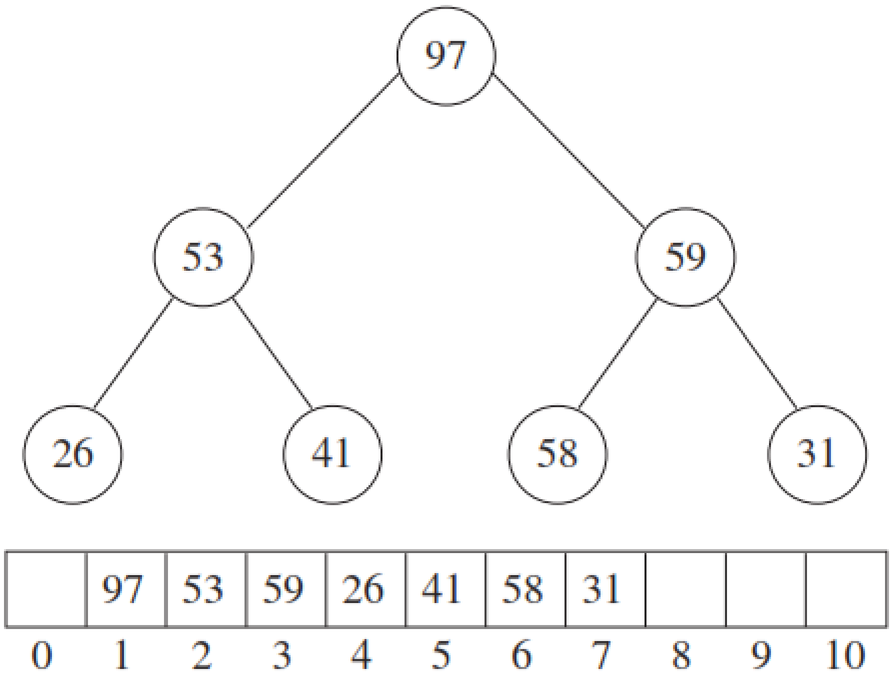
- priority queues can be used to sort in $O(N \lg N)$
- strategy
  - build binary heap of N elements – $O(N)$ time
  - perform N **`deleteMin`** operations – $O(N \lg N)$
  - elements (smallest first) stored in second array, then copied back into original array – $O(N)$ time
    - requires extra array, which doubles memory requirement

Heapsort

- can avoid extra array by storing element in original array
  - heap leaves an <u>open space</u> as each smallest element is deleted
  - store element in newly opened space, which is no longer used by the heap
  - results in list of <u>decreasing</u> order in array
  - to achieve increasing order, change ordering of heap to <u>max</u> heap
  - when complete, array contains elements in ascending order

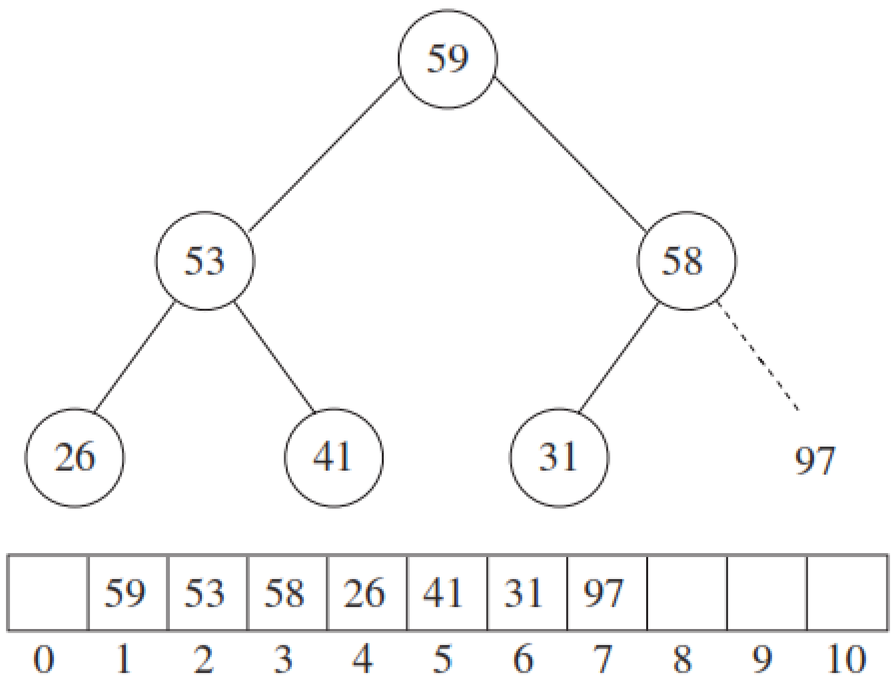
Heapsort

- max heap after **`buildHeap`** phase



Heapsort

- max heap after first **`deleteMax`**



Heapsort

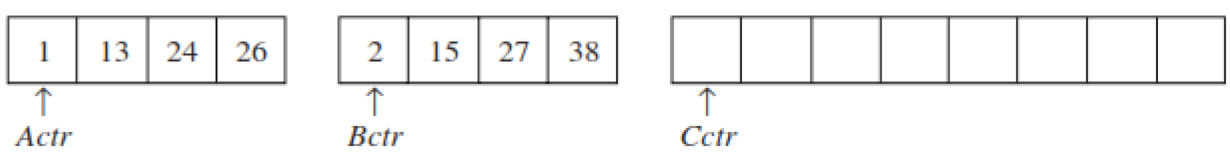
- analysis
  - building binary heap of N elements – < $2N$ comparisons
  - total **`deleteMax`** operations – $2N \lg N - O(N)$ if $N \geq 2$
  - heapsort in worst case – $2N \lg N - O(N)$
  - average case extremely complex to compute – $2N \lg N - O(N \lg \lg N)$
  - improved to $2N \lg N - O(N)$ or simply $O(N \lg N)$
- heapsort useful if we want to sort the <u>largest</u> k or <u>smallest</u> k elements and $k \ll N$

Mergesort

- mergesort is a divide-and-conquer algorithm
- in Vol. III of *The Art of Computer Programming*, Knuth attributes the algorithm to John von Neumann (1945)
- the idea of mergesort is simple:
  - divide the array in two
  - sort each half
  - merge the two subarrays using mergesort
    - merging simple since subarrays sorted
- mergesort can be implemented recursively and non-recursively
- runs in $O(N \lg N)$, worst case

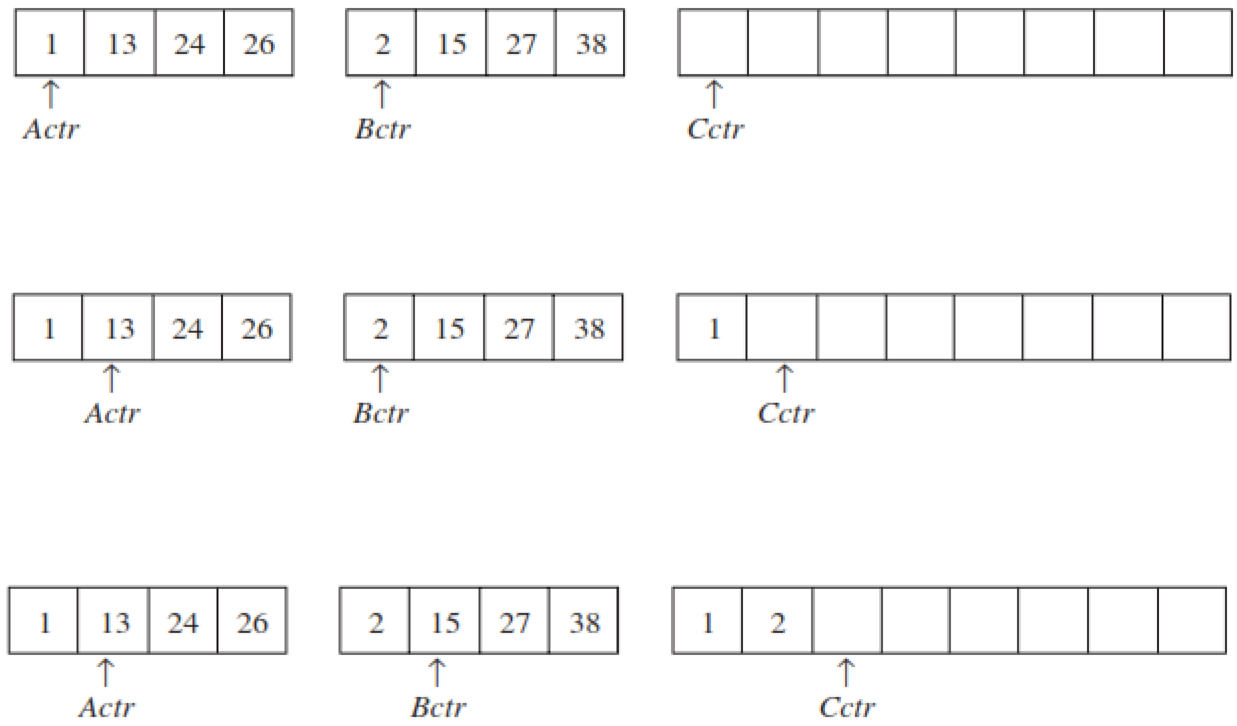
Mergesort

- merging algorithm takes two arrays, A and B, and <u>output</u> array C
  - also uses three counters: $Actr$, $Bctr$, $Cctr$
    - initialized to beginning of respective arrays
  - smaller of $A[Actr]$ and $B[Bctr]$ copied to $C[Cctr]$
  - when either input array exhausted, the remainder of the other list is <u>copied</u> to C



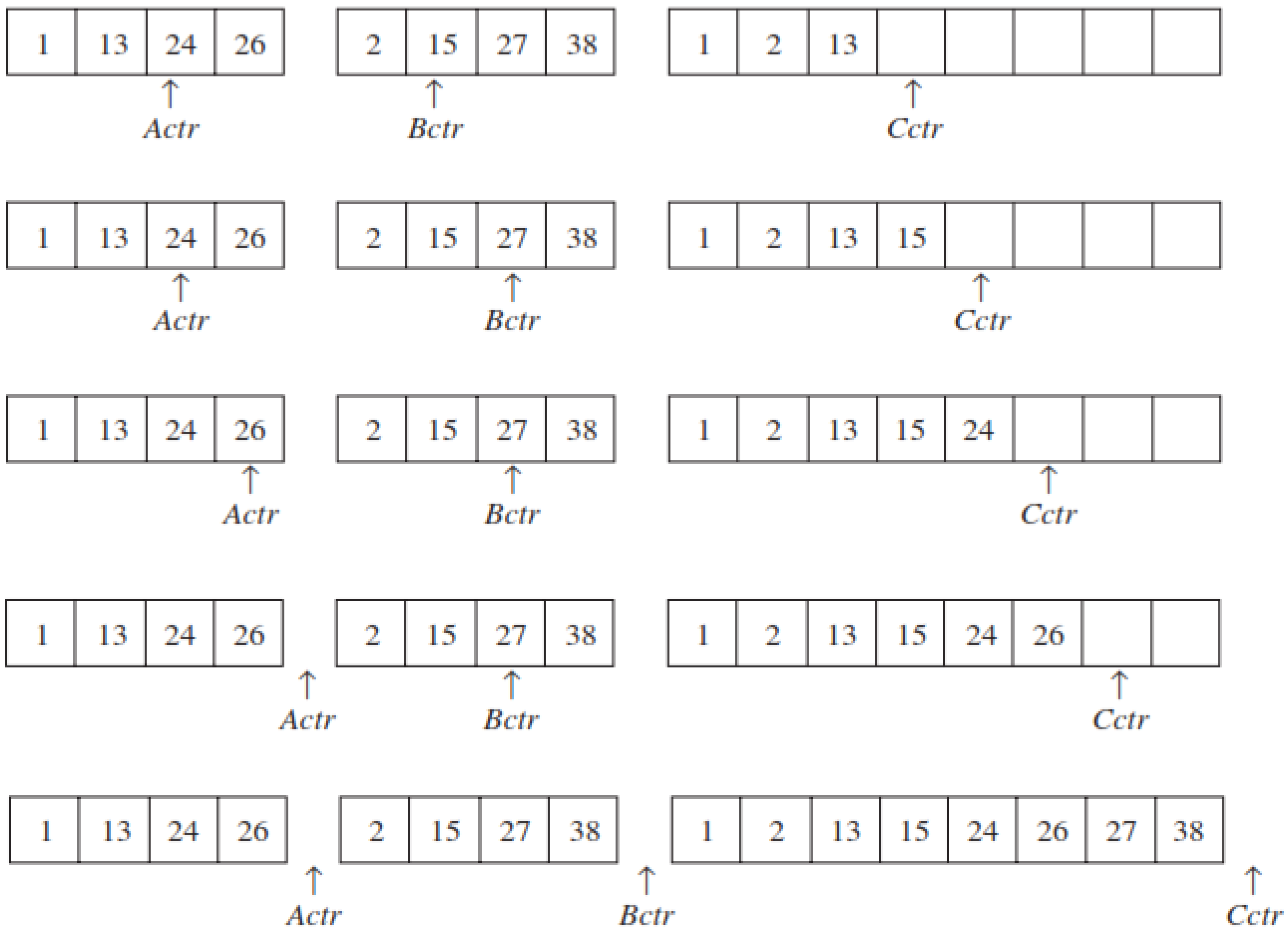
Mergesort

– example



Mergesort

– example (cont.)



Mergesort

- example (cont.)
  - another way to visualize

1	13	24	26		**2**	15	27	38									
1	**13**	24	26		**2**	15	27	38		1							
1	**13**	24	26		2	**15**	27	38		1	2						
1	13	**24**	26		2	**15**	27	38		1	2	13					
1	13	**24**	26		2	15	**27**	38		1	2	13	15				
1	13	24	**26**		2	15	**27**	38		1	2	13	15	24			
1	13	24	26		2	15	**27**	38		1	2	13	15	24	26		

Since there are no more elements from the left-half to process, we can simply copy all the remaining elements in the right-half:

1	13	24	26		2	15	27	38		1	2	13	15	24	26	27	38

Mergesort

- time to merge two lists is linear – at most $N - 1$ comparisons
  - every comparison adds an element to C
- mergesort easy to characterize
  - if $N = 1$, only one element to sort
  - otherwise, recursively mergesort first half and second half
  - merge these two halves
  - problem is *divided* into smaller problems and solved recursively, and *conquered* by patching the solutions together

Mergesort

- analysis
  - running time represented by recurrence relation
  - assume N is a power of 2 so that list is always divided evenly
  - for $N = 1$, time to mergesort is constant
  - otherwise, time to mergesort N numbers is time to perform two recursive mergesort of size $N/2$, plus the time to merge, which is linear

$$T(1) = 1$$

$$T(N) = 2T\left(\frac{N}{2}\right) + N$$

Mergesort

- analysis (cont.)
  - standard recurrence relation
  - can be solved in at least two ways
    - telescoping – divide the recurrence through by N
    - substitution

$$T(n) = \begin{cases} 2T(n/2) + c_1 n & \text{if } n > 1, \\ c_0 & \text{if } n = 1. \end{cases}$$

Mergesort

– solving mergesort recurrence relation using <u>telescoping</u>

$$T(n) = \begin{cases} 2T(n/2) + c_1 n & \text{if } n > 1, \\ c_0 & \text{if } n = 1. \end{cases}$$

For convenience, we assume $n = 2^k$ for some $k \geq 0$.

Note that

$$T(n)/n = T(n/2)/(n/2) + c_1,$$

so

$$T(n)/n - T(n/2)/(n/2) = c_1,$$

From the recursive nature of mergesort it follows that

$$\begin{aligned} T(n/2)/(n/2) - T(n/4)/(n/4) &= c_1 \\ T(n/4)/(n/4) - T(n/8)/(n/8) &= c_1 \\ \vdots &= \vdots \\ T(2)/2 - T(1) &= c_1 \end{aligned}$$

Mergesort

– solving mergesort recurrence relation using <u>telescoping</u> (cont.)

Adding up all these relations, we obtain

$$\begin{aligned}(T(n)/n - T(n/2)/(n/2)) + (T(n/2)/(n/2) - T(n/4)/(n/4)) \\ + (T(n/4)/(n/4) - T(n/4)/(n/8)) + \cdots + (T(2)/2 - T(1)) \\ = \underbrace{c_1 + c_1 + c_1 + \cdots + c_1}_{k \text{ times}}.\end{aligned}$$

After cancellation we are left with

$$\begin{aligned}T(n)/n - T(1) &= kc_1, \\ T(n) &= nT(1) + nkc_1, \\ T(n) &= c_0 n + c_1 n \lg n.\end{aligned}$$

Thus, the complexity of mergesort is $n \lg n$.

Mergesort

- solving mergesort recurrence relation using <u>substitution</u> (cont.)

Start with the general recurrence as Iteration 1:

$$T(n) = 2T(n/2) + c_1 n.$$

Iteration 2: since $T(n/2) = 2T(n/2^2) + c_1(n/2)$, we have

$$T(n) = 2(2T(n/2^2) + c_1(n/2)) + c_1 n = 2^2 T(n/2^2) + 2c_1 n.$$

Iteration 3: since $T(n/2^2) = 2T(n/2^3) + c_1(n/2^2)$, we have

$$T(n) = 2^2(2T(n/2^3) + c_1(n/2^2)) + 2c_1 n = 2^3 T(n/2^3) + 3c_1 n.$$

Iteration 4: since $T(n/2^3) = 2T(n/2^4) + c_1(n/2^3)$, we have

$$T(n) = 2^3(2T(n/2^4) + c_1(n/2^3)) + 3c_1 n = 2^4 T(n/2^4) + 4c_1 n.$$

At this point the pattern is clear: at the m-th iteration,

$$T(n) = 2^m T(n/2^m) + mc_1 n.$$

Mergesort

Choosing $m = k = \lg n$, we obtain

$$T(n) = 2^k T(n/2^k) + kc_1 n = nT(1) + c_1 n \lg n = c_0 n + c_1 n \lg n,$$

as before.

Mergesort

- proof by induction

  Prove: $T(n) = 2T\left(\frac{n}{2}\right) + c_1 n \quad T(1) = c_0$ is equivalent to

  $$T(n) = c_0 n + c_1 n \lg n$$

  let $n = 2^k$

  Base: $k = 0$ (or $n = 1$)

  rr: $T(1) = c_0$ by definition

  cf: $T(1) = c_0 \cdot 1 + c_1 \cdot 1 \cdot \lg 1$

  $= c_0 + c_1 \cdot 1 \cdot 0$

  $= c_0$ ✓

  I.H.: Assume: $T(2^k)$: $2T\left(\frac{2^k}{2}\right) + c_1 2^k = c_0 2^k + c_1 2^k \lg 2^k$

  for some $k \geq 1$

Mergesort

– proof by <u>induction</u> (cont.)

I.S.: Show: $2T\left(\frac{2^{k+1}}{2}\right) + c_1 2^{k+1} = c_0 2^{k+1} + c_1 2^{k+1} \lg 2^{k+1}$

$$
\begin{aligned}
2T\left(\frac{2^{k+1}}{2}\right) + c_1 2^{k+1} &= 2T\left(2^k\right) + c_1 2^{k+1} \\
&= 2\left(c_0 2^k + c_1 2^k \lg 2^k\right) + c_1 2^{k+1} \quad \text{by I.H.} \\
&= c_0 2^{k+1} + c_1 2^{k+1} \lg 2^k + c_1 2^{k+1} \\
&= c_0 2^{k+1} + c_1 2^{k+1}\left(\lg 2^k + 1\right) \\
&= c_0 2^{k+1} + c_1 2^{k+1}\left(\lg 2^k + \lg 2\right) \\
&= c_0 2^{k+1} + c_1 2^{k+1} \lg 2^{k+1} \quad \checkmark
\end{aligned}
$$

By induction, we have therefore shown the original statement to be true.

Quicksort

- historically, quicksort has been fastest known generic sorting algorithm
- average running time $O(N \lg N)$
- worst case running time $O(N^2)$, but can be made highly unlikely
- can be combined with heapsort to achieve $O(N \lg N)$ average and worst case time

Quicksort

- quicksort is a divide-and-conquer algorithm
- basic idea
  - arbitrarily select a single item
  - form three groups:
    - those <u>smaller</u> than the item
    - those <u>equal</u> to the item
    - those <u>larger</u> than the item
  - recursively sort the first and third groups
  - concatenate the three groups

Quicksort

```
template <typename Comparable>
void SORT( vector<Comparable> & items )
{
    if( items.size( ) > 1 )
    {
        vector<Comparable> smaller;
        vector<Comparable> same;
        vector<Comparable> larger;

        auto chosenItem = items[ items.size( ) / 2 ];

        for( auto & i : items )
        {
            if( i < chosenItem )
                smaller.push_back( std::move( i ) );
            else if( chosenItem < i )
                larger.push_back( std::move( i ) );
            else
                same.push_back( std::move( i ) );
        }

        SORT( smaller );     // Recursive call!
        SORT( larger );      // Recursive call!

        std::move( begin( smaller ), end( smaller ), begin( items ) );
        std::move( begin( same ), end( same ), begin( items ) + smaller.size( ) );
        std::move( begin( larger ), end( larger ), end( items ) - larger.size( ) );
    }
}
```

Quicksort

- implementation performance good on most inputs
- if list contains many duplicates, performance is very good
- some issues
  - making extra lists recursively consumes memory
    - not much better than mergesort
  - loop bodies too heavy
    - can avoid equal category in loop

Quicksort

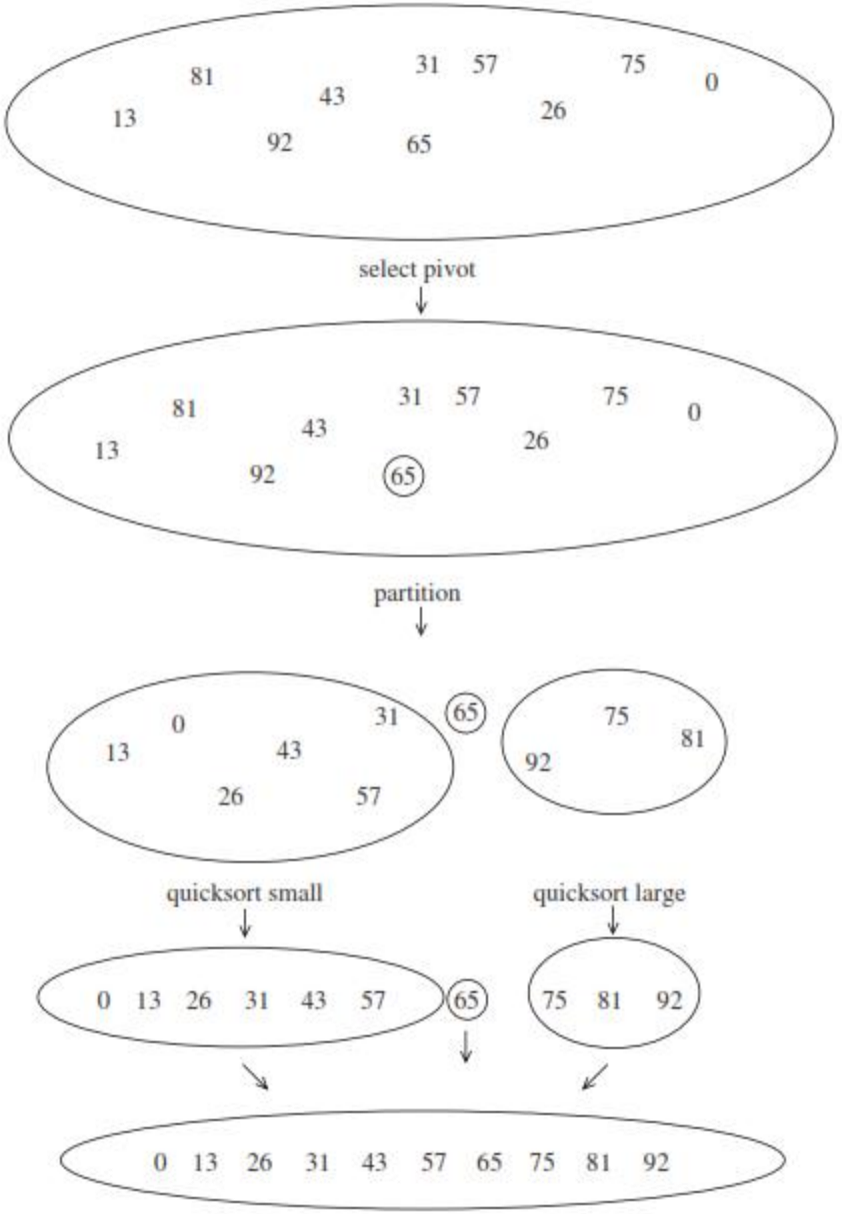
- classic quicksort algorithm to sort an array S
  - if there are either 0 or 1 elements in S, then <u>return</u>
  - choose an element v in S to serve as the <u>pivot</u>
  - partition $S - \{v\}$ into two disjoint subsets S_1 and S_2 with the properties that

    $x \leq v$ if $x \in S_1$ and

    $x \geq v$ if $x \in S_2$
  - apply quicksort recursively to S_1 and S_2
- note the <u>ambiguity</u> for elements equal to the pivot
  - ideally, half of the duplicates would go into each sublist

Quicksort

- example
  - pivot chosen randomly



Quicksort

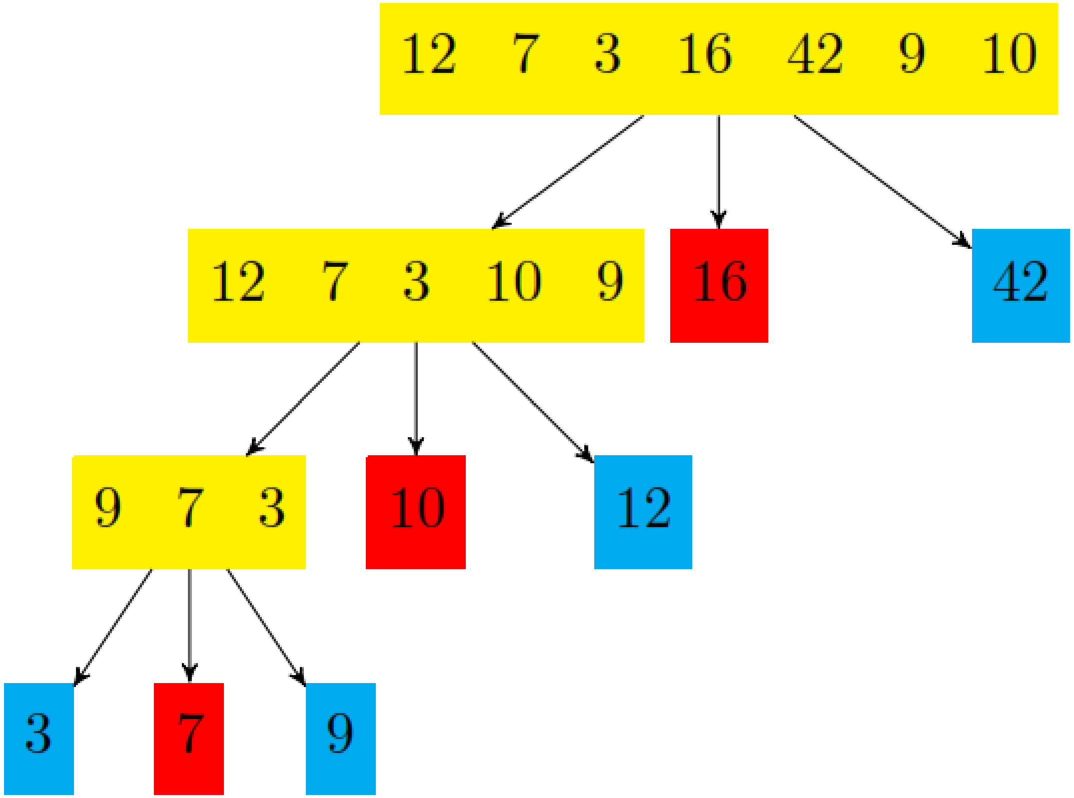
- many methods for selecting pivot and partitioning elements
  - performance very sensitive to even slight variances in these choices
- comparison with mergesort
  - like mergesort, recursively solves two subproblems and requires linear additional work
  - unlike mergesort, subproblems may not be of equal size (bad)

Quicksort

- quicksort vs. mergesort
  - mergesort: partitioning is trivial; the work is in the merge
  - quicksort: the work is in the partitioning; the merge is trival
  - mergesort: requires an auxiliary array to be efficient (in-place variants exist that are less efficient, or which sacrifice an important property called stability)
  - quicksort: faster since partitioning step can be performed efficiently in place (with a modest amount ($\lg N$) space needed to handle the recursion)
  - in both sorts, more efficient to switch to insertion sort once the arrays are sufficiently small to avoid the cost of the overhead of recursion on small arrays

Quicksort

- example
  - pivots in red



Quicksort

- choosing the pivot
  - popular, but bad, method: choose the <u>first</u> element in the list
    - OK if input is <u>random</u>
    - not OK if input is presorted or in reverse order
      - happens consistently in recursive calls
      - results in <u>quadratic</u> time for presorted data for doing nothing!
      - occurs often
      - alternative: choose larger of first two elements
  - could pick the pivot randomly
    - safe, but random number generation expensive

Quicksort

- choosing the pivot (cont.)
  - median-of-three partitioning
    - best choice would be median of sublist, but takes too long to calculate
    - good estimate by picking three elements <u>randomly</u> and using middle element as pivot
      - randomness not really helpful
      - select first, middle, and last elements
    - eliminates bad case for <u>sorted</u> input
    - reduces number of <u>comparisons</u> by about 15%
  - example: 8, 1, 4, 9, 6, 3, 5, 2, 7, 0
    - from 8, 0, and $\lfloor (left + right)/2 \rfloor$, or 6, select 6

Quicksort

- partitioning strategy
  - first, get pivot out of the way by swapping with <u>last</u> element
  - two counters, *i* and *j*
    - *i* starts at first element
    - *j* starts at next-to-last element

8	1	4	9	0	3	5	2	7	6
↑								↑	
i								j	

  - move all the smaller elements to <u>left</u> and all larger elements to <u>right</u>

Quicksort

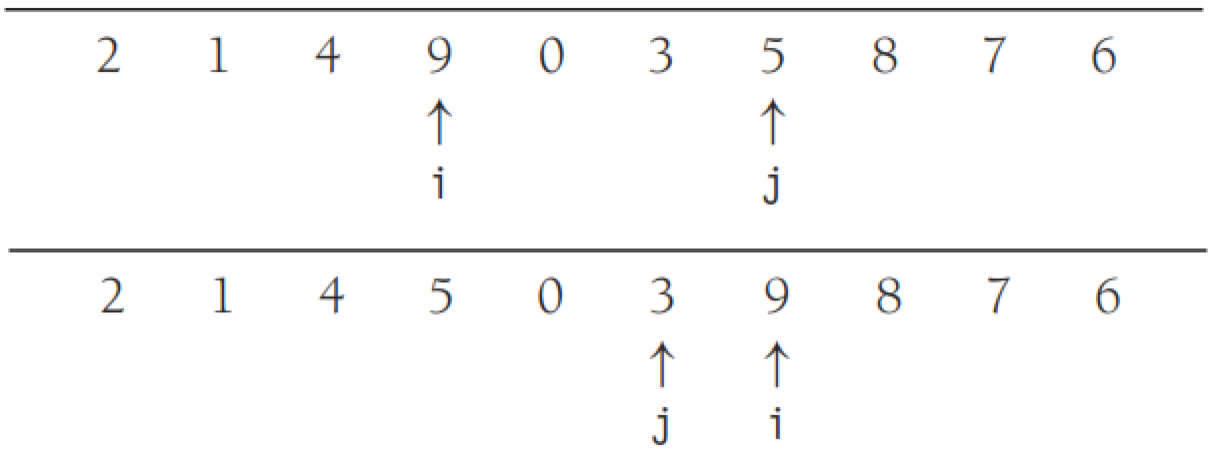
- partitioning strategy (cont.)
  - while i is to the left of j
    - move i right, skipping over elements smaller than pivot
    - move j left, skipping over elements larger than pivot
  - when i and j stop, i is at a larger element and j is at a smaller element – swap them
  - example

8	1	4	9	0	3	5	2	7	6
↑							↑		
i							j		

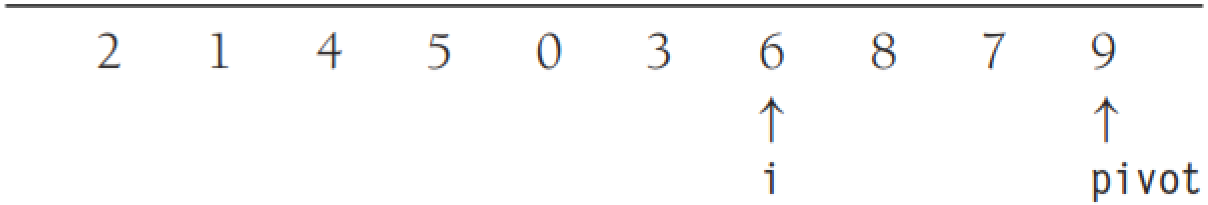
2	1	4	9	0	3	5	8	7	6
↑							↑		
i							j		

Quicksort

- partitioning strategy (cont.)
  - example (cont.)



  - after i and j cross, swap location i with pivot



Quicksort

- partitioning strategy (cont.)
  - at this point, all positions $p < i$ contain smaller elements than pivot, and all positions $p > i$ contain larger elements
  - how to handle equal elements
    - should i stop when element equal to pivot? what about j?
    - i and j should behave similarly to avoid all elements equal to pivot collecting in one sublist
    - best to have i and j stop and perform an unnecessary swap to avoid uneven sublists (and quadratic run time!)
  - for small arrays, and as sublists get small (< 20 elements), use insertion sort
    - fast and avoids degenerate median-of-three cases

Quicksort

- implementation
  - driver

```
/**
 * Quicksort algorithm (driver).
 */
template <typename Comparable>
void quicksort( vector<Comparable> & a )
{
    quicksort( a, 0, a.size( ) - 1 );
}
```

  - pass array and range (left and right) to be sorted

Quicksort

- implementation (cont.)
  - median-of-three pivot selection
    - sort **`a[left]`**, **`a[right]`**, and **`a[center]`** in <u>place</u>
    - smallest of three ends up in <u>first</u> location
    - largest in last location
    - pivot in **`a[right – 1]`**
    - i can be initialized to **`left + 1`**
    - j can be initialized to **`right – 2`**
    - since **`a[left]`** smaller than pivot, it will act as a <u>sentinel</u> and stop j from going past the beginning of the array
    - storing pivot at **`a[right – 1]`** will act as a sentinel for i

Quicksort

- implementation (cont.)
  - median-of-three

```
/**
 * Return median of left, center, and right.
 * Order these and hide the pivot.
 */
template <typename Comparable>
const Comparable & median3( vector<Comparable> & a, int left, int right )
{
    int center = ( left + right ) / 2;

    if( a[ center ] < a[ left ] )
        std::swap( a[ left ], a[ center ] );
    if( a[ right ] < a[ left ] )
        std::swap( a[ left ], a[ right ] );
    if( a[ right ] < a[ center ] )
        std::swap( a[ center ], a[ right ] );

        // Place pivot at position right - 1
    std::swap( a[ center ], a[ right - 1 ] );
    return a[ right - 1 ];
}
```

Quicksort

- implementation (cont.)
  - main quicksort

```
/**
 * Internal quicksort method that makes recursive calls.
 * Uses median-of-three partitioning and a cutoff of 10.
 * a is an array of Comparable items.
 * left is the left-most index of the subarray.
 * right is the right-most index of the subarray.
 */
template <typename Comparable>
void quicksort( vector<Comparable> & a, int left, int right )
{
    if( left + 10 <= right )
    {
        const Comparable & pivot = median3( a, left, right );

```

Quicksort

- implementation (cont.)
  - main quicksort (cont.)

```
            // Begin partitioning
        int i = left, j = right - 1;
        for( ; ; )
        {
            while( a[ ++i ] < pivot ) { }
            while( pivot < a[ --j ] ) { }
            if( i < j )
                std::swap( a[ i ], a[ j ] );
            else
                break;
        }

        std::swap( a[ i ], a[ right - 1 ] );  // Restore pivot

        quicksort( a, left, i - 1 );     // Sort small elements
        quicksort( a, i + 1, right );    // Sort large elements
    }
    else  // Do an insertion sort on the subarray
        insertionSort( a, left, right );
}
```

Quicksort

- implementation (cont.)
  - main quicksort (cont.)
    - 16: **`i`** and **`j`** start at one off
    - 22: swap can be written inline
    - 19-20: small inner loop very fast

Quicksort

- analysis
  - quicksort is interesting because its worst-case behavior and its expected behavior are very different
  - let $T(n)$ be the run-time needed to sort n items
    - $T(0) = T(1) = 1$
    - pivot selection is constant time
    - cost of the partition is cn
    - if S_1 has i elements, then S_2 has $n - i - 1$ elements, and

$$T(n) = T(i) + T(n - i - 1) + cn$$

Quicksort

- worst-case analysis
  - the worst-case occurs when $i = 0$ or $i = n$ i.e., when the pivot is the smallest or largest element every time quicksort() is called
  - in this case, without loss of generality we may assume that $i = 0$, so
    $T(n) = T(0) + T(n-1) + cn \sim T(n-1) + cn, n > 1$
  - thus

$$
\begin{aligned}
T(n-1) &= T(n-2) + c(n-1) \\
T(n-2) &= T(n-3) + c(n-2) \\
T(n-3) &= T(n-4) + c(n-3) \\
&\ldots \\
T(3) &= T(2) + c(3) \\
T(2) &= T(1) + c(2)
\end{aligned}
$$

Quicksort

- worst-case analysis (cont.)
  - combining these yields

$$T(n) = cn + c(n-1) + c(n-2) + \ldots + (c \times 3) + (c \times 2) + T(1)$$

  - or

$$T(n) = T(1) + c\sum_{k=2}^{n} k \sim c\frac{n^2}{2}$$

  - quadratic!
    - is it likely that at every recursive call to **`quicksort()`** we will choose the smallest element as the pivot?
      - yes, if the data are already sorted

Quicksort

- best-case analysis
  - in the best case, the pivot is always the median of the data being operated on

$$T(n) = T(n/2) + T(n/2) + cn = 2T(n/2) + cn$$

  - we know from the analysis of mergesort that the solution is

$$T(n) = \Theta(n \lg n)$$

Quicksort

- average-case analysis
  - assumption: any partition size is equally likely
  - for instance, suppose $n = 7$; since we remove the pivot, the possible sizes of the partitions are

	Size of partition						
S_1	0	1	2	3	4	5	6
S_2	6	5	4	3	2	1	0

  - in this case the expected value of $T(i) + T(n - i - 1)$ is

$$\frac{2}{6}\sum_{j=0}^{6} T(j)$$

Quicksort

In general, the expected complexity of quicksort is

$$T(n) = cn + \frac{2}{n}\sum_{j=0}^{n-1} T(j) = cn + \frac{2}{n}(T(0) + T(1) + \cdots T(n-1)),$$

whence

$$nT(n) = cn^2 + 2(T(0) + T(1) + \cdots T(n-1)).$$

The same reasoning tells us that

$$(n-1)T(n-1) = c(n-1)^2 + 2(T(0) + T(1) + \cdots T(n-2)).$$

Thus,

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= c(2n-1) + 2T(n-1) \\ nT(n) &= (n+1)T(n-1) + c(2n-1) \end{aligned}$$

Quicksort

We may safely ignore a $-c$ term:

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

This last recursion yields

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

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1} \qquad \Rightarrow \frac{T(n)}{n+1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1}$$

$$\vdots = \vdots$$

$$\frac{T(3)}{4} = \frac{T(2)}{3} + \frac{2c}{4}$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \qquad \Rightarrow \frac{T(n)}{n+1} = \frac{T(1)}{2} + \frac{2c}{3} + \cdots + \frac{2c}{n+1}.$$

Quicksort

Thus we have arrived at

$$\begin{aligned}\frac{T(n)}{n+1} &= \frac{T(1)}{2} + 2c\left(\frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n+1}\right)\\ &= \frac{T(1)}{2} + 2c\left(\left(\sum_{k=1}^{n+1}\frac{1}{k}\right) - \frac{1}{1} - \frac{1}{2}\right)\\ &\approx \frac{T(1)}{2} + 2c\left(\ln(n+1) + \gamma - \frac{3}{2}\right)\end{aligned}$$

and thus

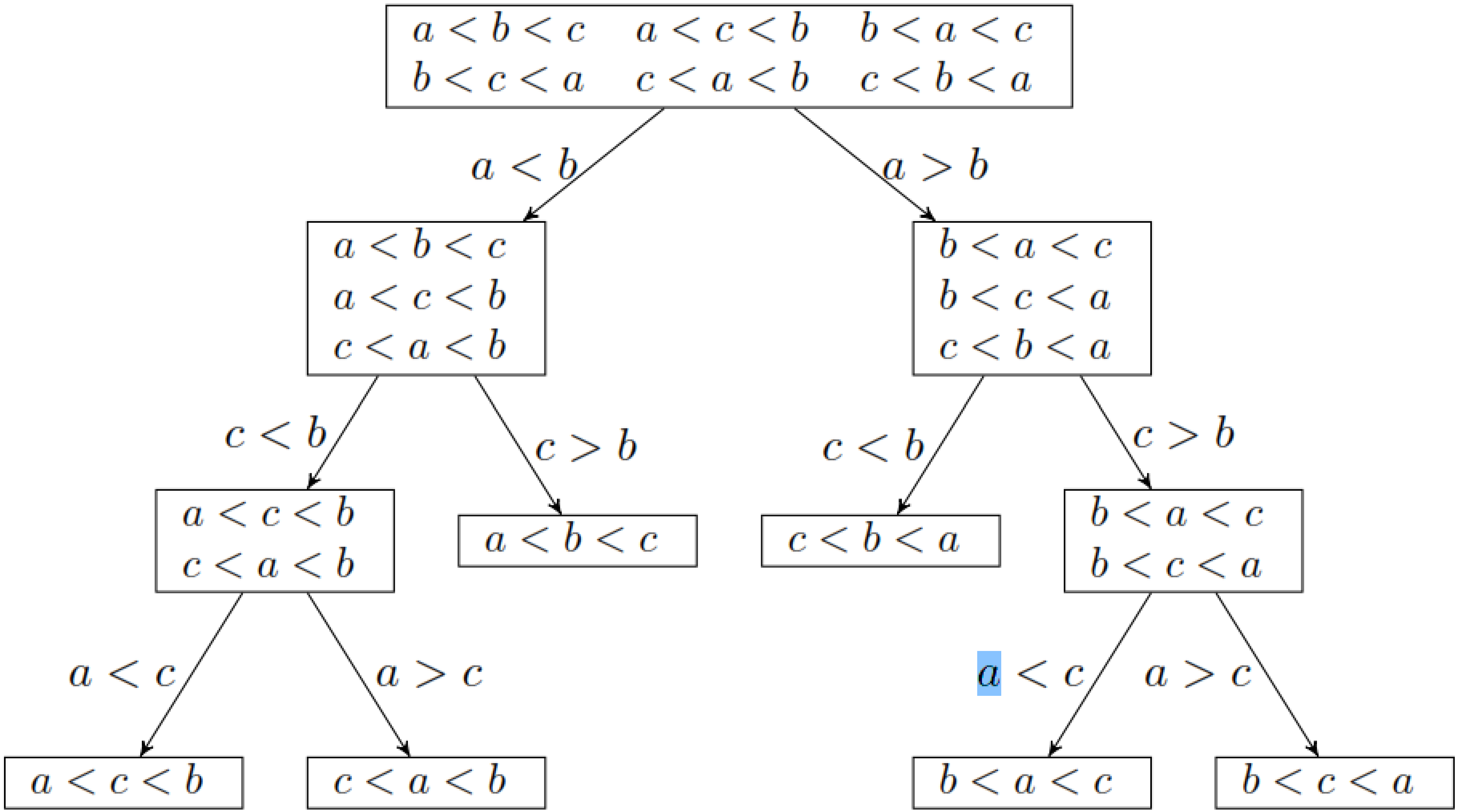
$$T(n) \sim 2cn\ln n.$$

The expected behavior is comparable to the best-case behavior!

Lower Bound for Pairwise Sorting

- no algorithm based on pairwise comparisons can guarantee sorting n items with fewer than $\lceil \lg n! \rceil \sim n \lg n$ comparisons
  - to show this, we first abstract the behavior of such algorithms using a decision tree
  - a decision tree is a binary tree in which each node represents a set of possible orderings
  - the root consists of the $n!$ possible orderings of the items to be sorted
  - the edges represent the results of comparisons, and a node comprises the orderings consistent with the comparisons made on the path from the root to the node
  - each leaf consists of a single sorted ordering

Lower Bound for Pairwise Sorting



Lower Bound for Pairwise Sorting

- a decision tree to sort n items must have $n!$ leaves
  - this requires a tree of depth $\lceil \lg n! \rceil \sim n \lg n$ by Stirling's approximation
  - thus, the best case for sorting with pairwise comparisons is $\Omega(n \lg n)$

Quickselect

- thus far, the best performance to select the k^{th} smallest element is $O(N \lg N)$ using a priority queue (heap)
- quicksort can be modified to solve the selection problem
  - quickselect

Quickselect

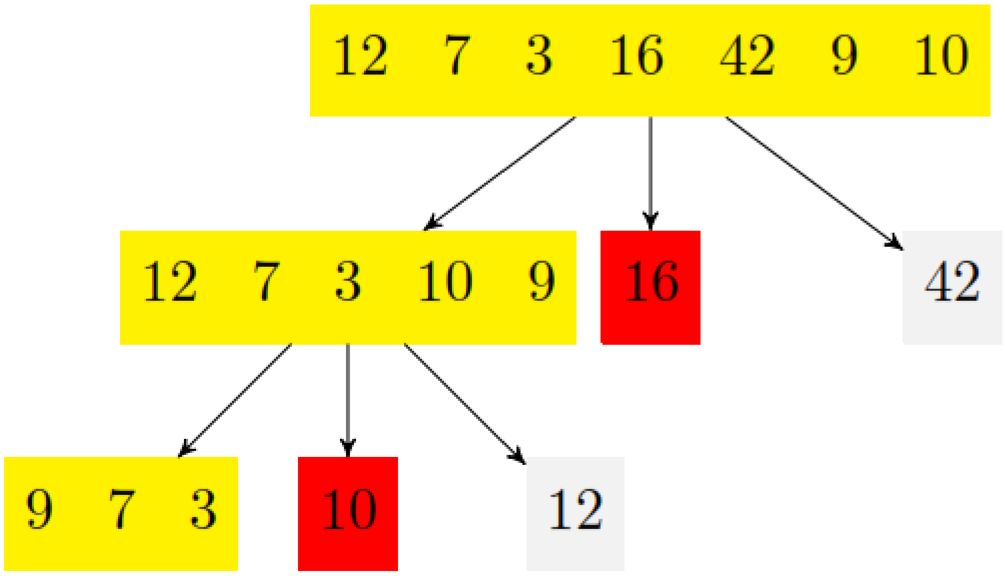
- quickselect algorithm
  - given a set S, let $|S|$ be its cardinality
  - quickselect (S, k)
    - if there is 1 element in S, return $k = 1$
    - choose an element in S to serve as the pivot
    - partition $S - \{v\}$ into two disjoint subsets S_1 and S_2 with the properties that

      $x \leq v$ if $x \in S_1$ and

      $x \geq v$ if $x \in S_2$
    - now the search proceeds on S_1 and S_2
      - if $k \leq |S_1|$, then the k^{th} smallest element must be in S_1, so quickselect(S_1,k)
      - if $k = 1 + |S_1|$, then the pivot is the k^{th} smallest, so return v
      - otherwise, the k^{th} smallest element must be in S_2, and it is the $(k - |S_1| - 1)$-th element of S_2, so return quickselect(S_2, $k - |S_1| - 1$)

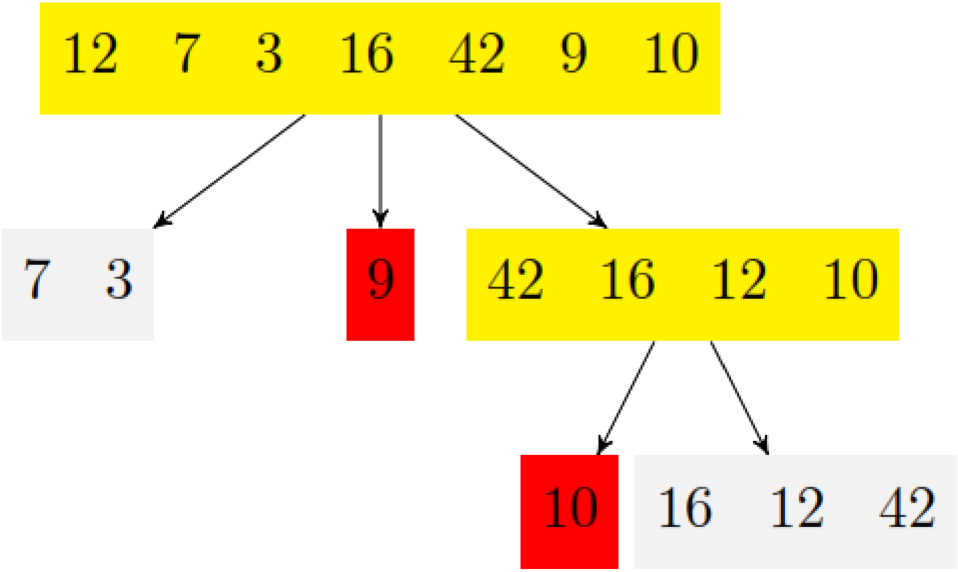
Quickselect

- example: find the median of 7 items ($k = 4$)
  - red denotes pivots, while grey denotes the partition that is <u>ignored</u>
  - call quickselect $(S, 4)$; partition, then call quickselect $(S_1, 4)$; once again, partition; at this point, $|S_1| = 3$, so the pivot is the 4th element, and thus the answer



Quickselect

- example: find the median of 7 items ($k = 4$)
  - call quickselect ($S, 4$); partition; since $|S_1| = 2$, we want the $k - |S_1| - 1 = 4 - 1 - 1 = 1st$ smallest element of S_2 , so call quickselect (S_2,1); partition; since we are inside the call quickselect (S_2,1), we want the 1st smallest element, so we call quickselect (S_1,1), which immediately exits, returning 10



Quickselect

- quickselect complexity
  - at each recursive step quickselect ignores one partition – will this make it faster than quicksort?
  - in the worst case, quickselect behaves like quicksort, and has n^2 complexity
    - this occurs if the one partition is <u>empty</u> at each partitioning, and we have to look at all the terms in the other partition.
  - best case behavior is <u>linear</u>
    - occurs if each partition is <u>equal</u>
    - since quickselect ignores one partition at each step, its runtime $T(n)$ satisfies the recurrence

$$T(n) = T\left(\frac{n}{2}\right) + cn$$

    - this leads to $T(n)$ being linear

Quickselect

- quickselect complexity (cont.)
  - expected behavior
    - suppose we choose our pivot v randomly from the terms we are searching
    - suppose v lies between the 25th and 75th percentiles of the terms (i.e., v is larger than 1/4 and smaller than 1/4 of the terms)
    - this means that neither partition can contain more than 3/4 of the terms, so the partitions can't be too imbalanced; call such a pivot "good"
    - on average, how many v do we need to choose before we get a good one?
      - a randomly chosen v is good with probability ½ - a good pivot lies in the middle of 50% of the terms
      - choosing a good pivot is like tossing a coin and seeing heads

Quickselect

- quickselect complexity (cont.)
  - expected behavior (cont.)
    - the expected number of tosses to see a heads is two
    - to see this, let E be the expected number of tosses before seeing a heads
    - toss the coin; if it's heads, we're done; if it's tails (which occurs with probability 1/2) we have to toss it again, so

$$E = 1 + \frac{1}{2}E, \quad \text{whence } E = 2$$

    - thus, on average, quickselect will take two partitions to reduce the array to at most 3/4 of the original size

Quickselect

- quickselect complexity (cont.)
  - expected behavior (cont.)
    - in terms of $T(n)$,
      - expected value of $T(n) \leq T(3n/4) +$ expected time to reduce the array
    - since each partitioning step requires cn work, and we expect to need 2 of them to reduce the array size to $\leq 3n/4$, we have

$$T(n) \leq T(3n/4) + cn$$

Quickselect

- quickselect complexity (cont.)
  - expected behavior (cont.)
    - consider the more general recurrence

$$T(n) \leq T(\alpha n) + cn, \text{ where } \alpha < 1$$

    - at the k^{th} level of the recursion, starting with $k = 1$, there is a single problem of size at most $\alpha^k n$
    - the amount of work done at each level is thus at most $c\alpha^k n$
    - the recursion continues until

$$\alpha^m n \leq 1$$

    - so $m \log \alpha + \log n \leq 0$, or

$$m \leq -\frac{\log n}{\log \alpha}$$

Quickselect

- quickselect complexity (cont.)
  - expected behavior (cont.)
    - thus, the total amount of work is bounded above by

$$cn + c\alpha n + c\alpha^2 n + \cdots + c\alpha^{\lceil m \rceil} n = cn\frac{1 - \alpha^{\lceil m+1 \rceil}}{1 - \alpha} \leq cn\frac{1}{1 - \alpha} = O(n)$$

    - thus, the best case and expected case behavior of quickselect with a randomly chosen pivot is $O(n)$

Introsort

- from D. R. Musser, Introspective sorting and selection algorithms, Software: Practice and Experience 27 (8) 983-993, 1997
- introsort is a <u>hybrid</u> of quicksort and heapsort
- introsort starts with quicksort, but switches to heapsort whenever the number of levels of recursion exceed a specified threshold (e.g., $2\lfloor \lg n \rfloor$).
- if it switches to heapsort, the subarrays that remain to be sorted will likely be much <u>smaller</u> than the original array
- this approach gives an $n \lg n$ <u>guaranteed</u> complexity, while making the most of the efficiency of quicksort

Stability

- a sorting algorithm is stable if it preserves the relative order of <u>equal</u> keys
- stable sorts:
  - insertion sort
  - mergesort
- unstable sorts:
  - selection sort
  - Shell sort
  - quicksort
  - heapsort

Stability

Original list (sorted by title)

Austen	Emma
Shakespeare	Hamlet
Shakespeare	King Lear
Shakespeare	Macbeth
Austen	Pride and Prejudice
Shakespeare	Romeo and Juliet
Austen	Sense and Sensibility

Unstably sorted by author

Austen	Pride and Prejudice
Austen	Emma
Austen	Sense and Sensibility
Shakespeare	Romeo and Juliet
Shakespeare	Hamlet
Shakespeare	Macbeth
Shakespeare	King Lear

Stably sorted by author

Austen	Emma
Austen	Pride and Prejudice
Austen	Sense and Sensibility
Shakespeare	Hamlet
Shakespeare	King Lear
Shakespeare	Macbeth
Shakespeare	Romeo and Juliet

C++ Standard Library Sorts

- **`sort()`**: introsort, guaranteed to be $n \lg n$
- **`stable_sort()`**: mergesort, guaranteed to be stable and
  - $n \lg n$ time, if a $\Theta(n)$-sized auxiliary array can be allocated
  - $n \lg^2 n$ time for an in-place sort, otherwise
- **`partial_sort()`**: sort the k largest (or smallest) items

String / Radix Sort

- we can use the special structure of character strings to devise sorting algorithms
- here, a character should be understood in a general sense, as an element of an alphabet, which is a collection of M items (presumably all requiring the same number of bits for their representation)
- the alphabet is also assumed to have an ordering, so we may sort characters
- as a special case, we can think of treating base-b numbers as a string of digits from the range 0 to $b^p - 1$ (p-digit base-b numbers)
- since the base of a number system is sometimes called the radix, the sorting algorithms we will discuss are frequently called radix sorts

String Sorting: LSD

– in least-significant digit (LSD) first sorting, we sort using the digits (characters) from least- to most-significant:

initial	sorted by 1's	sorted by 10's	sorted by 100's
064	000	000	000
008	001	001	001
216	512	008	008
512	343	512	027
027	064	216	064
729	125	125	125
000	216	027	216
001	027	729	343
343	008	343	512
125	729	064	729

String Sorting: LSD

- algorithm:

```
for d from least- to most-significant digit {
   sort using counting sort on the least-significant digit
}
```

- observations:
  - since counting sort is stable, LSD sorting is stable
    - the stability of counting sort is essential to making LSD sorting work
  - algorithm assumes all the strings have the same length p
  - time complexity: there are p passes in the outmost loop; in each loop iteration, we apply counting sort to N digits in the range 0 to $b - 1$, requiring $N + b$; total work: $\Theta(p(N + b))$
  - space complexity: $N + b$ – same as counting sort

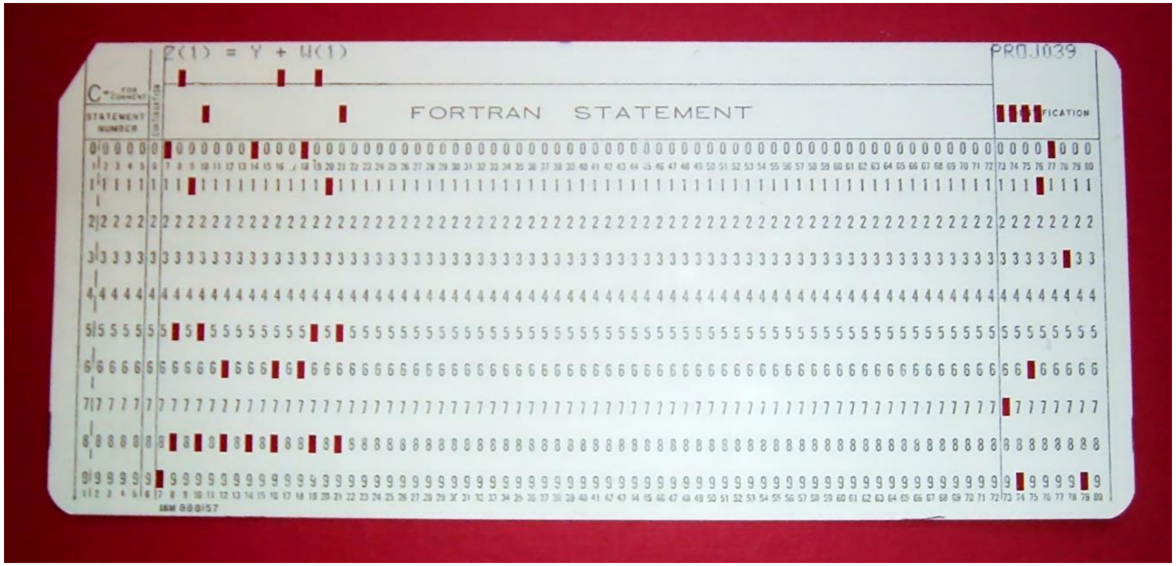
String Sorting: LSD

- why not go left-to-right?
  - the same idea, but starting with the most significant digit, doesn't work:

initial	sorted by 10's	sorted by 1's
21	12	21
12	21	12

- why does right-to-left digit sorting work but left-to-right does not?

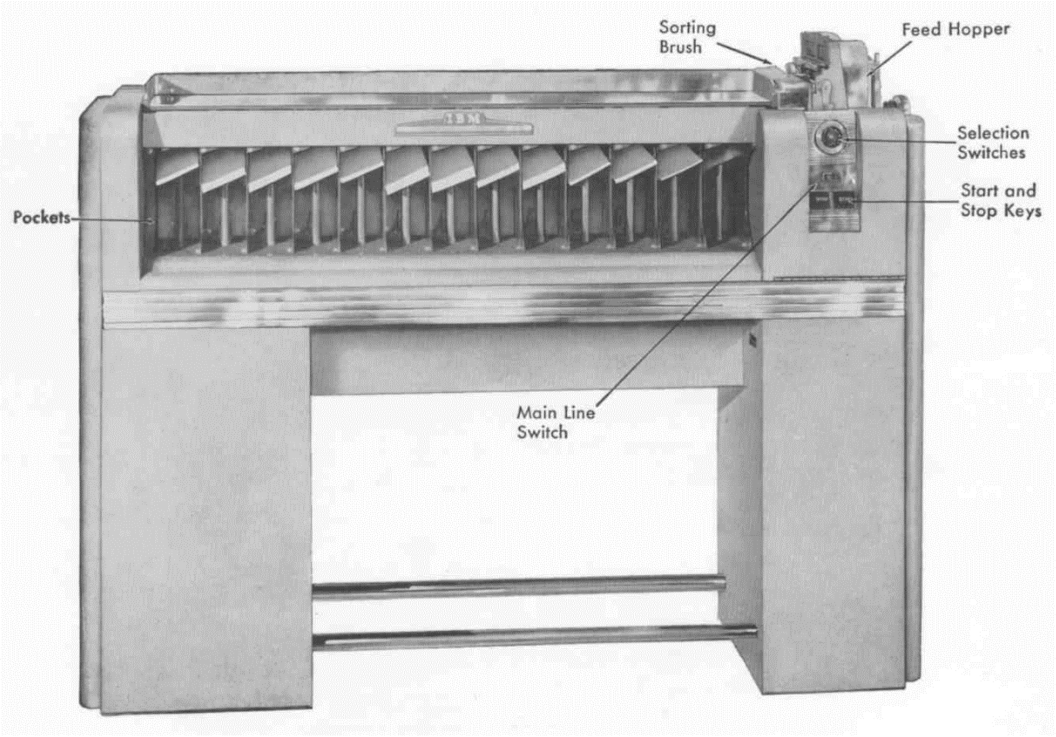
- in the old days, we used Hollerith cards:



- early versions of <u>Fortran</u> and COBOL had limits of 72 characters per line
- this left columns 73-80 free for the card <u>sequence</u> number
- that way, if you dropped your deck of cards, you could go to...

String Sorting: LSD

– … the card sorter



IBM Model 082 card sorter

String Sorting: LSD

- these machines used LSD:
  - pile the disordered cards in the input hopper and sort by the <u>last</u> digit in the sequence number
  - now take the sorted decks and stack them in order
  - place the combined deck back in the input hopper and sort by the next-to-last digit in the sequence number
  - repeat steps 2 and 3 until sorted

Sorting Algorithms Summary

algorithm	stable?	in place?	order of growth to sort n items: running time	extra space
selection sort	no	yes	n^2	1
insertion sort	yes	yes	n (best-case) n^2 (worst-case)	1
shellsort	no	yes	$< n^2$ ($n^{4/3}$ to $n^{3/2}$)	1
mergesort	yes	no	$n \lg n$	n
quicksort	no	yes	$n \lg n$ (expected) n^2 (worst-case)	$\lg n$
heapsort	no	yes	$n \lg n$	1
introsort	no	yes	$n \lg n$	$\lg n$

- Shell sort is subquadratic with a suitable increment sequence
  - Shell's original increment sequence is, in fact, quadratic in the worst case