Iterative binary searching

```
int bsearch(Type key, Type a[], int n) {
   int low = 0, high = n-1, middle;
   while (low <= high) {
      middle = (low + high) / 2;
      if (key == a[middle])
         return middle; /* success */
      if (key > a[middle]) low = middle + 1;
      else high = middle – 1;
   }
   return -1; /* unsuccessful */
}
```

- Both versions take $\log_2 n$ steps on average to find a value or find out the value is not in the array

Towers of Hanoi and 8 Queens

- Move n disks from a to c; use b to hold (demo)
  `void tower(int n, int a, int b, int c)`
  - Base case: just one disk – trivial
    `if (n==1) moveOneDisk(a→c);`
  - General case: *assume* a method that can move a tower of height n-1. This method!!!

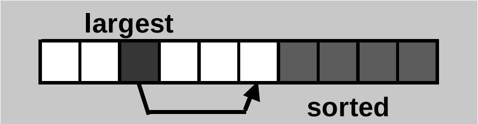
```
else {
    tower(size n-1, a→b with c holding);
    moveOneDisk(a→c);
    tower(size n-1, b→c with a holding);
}
```

- One more example – 8 queens problem

Sorting

- Probably *the* most expensive common operation
  - And maybe the most studied CS problem
- Problem: arrange `a[0..n-1]` by some ordering
  - e.g., in ascending order: `a[i-1]<=a[i], 0<i<n`
- Two general types of strategies
  - Comparison-based sorting – includes most strategies
    - Lots of simple, inefficient algorithms
    - Some not-so-simple, but more efficient algorithms
  - Address calculation sorting – rarely used in practice
    - But very fast if the data are suitable

Selection sort

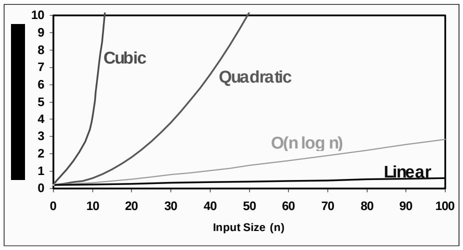


- Idea: build sorted sequence at end of array
- At each step:
  - *Find largest* value in not-yet-sorted portion
  - *Exchange* this value with the one at end of unsorted portion (now beginning of sorted portion)
- Easy to do (see text p. 629), but complexity is $O(n^2)$
  - Huh?

Big-Oh notation

- A way to compare algorithms – *just* algorithms
- All but the “dominant” term are ignored
  - e.g., say algorithm takes $3n^2 + 15n + 100$ steps (problem of size `n`) – 1st term dominates for large `n`
- Constants are due to processor speed, compiler, language features, … – so ignore the `3`
- Means this example algorithm is $O(n^2)$
  - Pronounced “Oh of n-squared” – a.k.a., it is in the “quadratic complexity” class of algorithms

Some complexity classes



- Linear - $O(n)$; Quadratic - $O(n^2)$; Cubic - $O(n^3)$
  - Also slower than cubic – e.g., Exponential - $O(2^n)$
  - And faster than linear – `O(log n)`, and Constant - `O(1)`

mergeSort

- A “divide and conquer” sorting strategy
  - Idea: (1) divide array in two; (2) sort each part; (3) combine two parts to overall solution
- mergeSort – has a naturally recursive solution

```
if (more than one item in array):
   divide array into left half and right half;
   mergeSort(left half); mergeSort(right half);
   merge(left half and right half together);
```

  - Requires helper method to merge two halves
    - Actually where all the work is done (p. 640)
- Complexity is `O(n log n)`
  - i.e., *lots faster* than selectionSort

How much faster is *lots faster*?

- Use a stopwatch to get some idea
  - See SelectionSortTimer
  - Of course – actual times depend on …
  - But MergeSortTimer is clearly much faster
- Moral: sometimes it pays to apply a better algorithm – despite the extra effort.

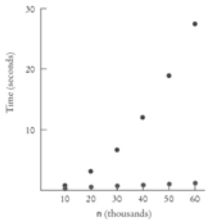


Figure 2
Merge Sort Timing (blue) versus Selection Sort (red)