CSE 142
Programming I

Sorting

Sorting

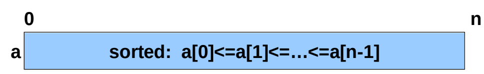
- The problem: put things in order
  - Usually smallest to largest: "ascending"
  - Could also be largest to smallest: "descending"
- More formally:
  - Given an array a[0], a[1], ... a[n-1], reorder entries so that a[0] <= a[1] <= ... <= a[n-1]
- Shorthand for these slides: the notation *array[i..k]* means all of the elements *array[i],array[i+1]...array[k]*
  - This is not C syntax!
- The array above would then be a[0..n-1]

Sorting

- Lots of applications
  - ordering hits in web search engine
  - preparing lists of output
  - merging data from multiple sources
  - to help solve other problems
    - faster search (allows binary search)
  - too many to mention!
- Sorting has been intensively studied for decades
- Many different ways to do it! We'll look at two algorithms
  - More in CSE143, CSE373, CSE326...

Sorting Problem

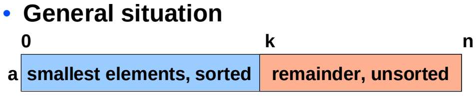
- **What we want: Data sorted in order**



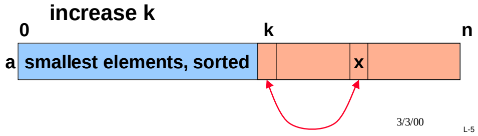
- **Initial conditions**

"Selection Sort"

- **General situation**



- **Step:**
  - **Find smallest element x in a[k..n-1]**
  - **Swap smallest element with a[k], then increase k**



Subproblem: Find Smallest

```
/* Yield location of smallest element in a[k..n-1] */
/* Assumption: k < n */
/* Returns index of smallest, does not return the
   smallest value itself */
int min_loc (int a[ ], int k, int n) {
   int j, pos;  /* a[pos] is smallest element */
                /* found so far */
   pos = k;
   for ( j = k + 1; j < n; j = j + 1)
     if (a[j] < a[pos])
       pos = j;
   return pos;
}
```

Code for Selection Sort

```
/* Sort a[0..n-1] in non-decreasing order (rearrange
   elements in a so that a[0]<=a[1]<=…<=a[n-1] ) */
int sel_sort (int a[ ], int n) {
   int k, m;
   for (k = 0; k < n - 1; k = k + 1) {
      m = min_loc(a,k,n);
      swap(&a[k], &a[m]);
   }
}
```


Example

a	3	12	-5	6	142	21	-17	45

a	-17	12	-5	6	142	21	3	45

a	-17	-5	12	6	142	21	3	45

Example (cont)

a	-17	-5	3	6	142	21	12	45

a	-17	-5	3	6	142	21	12	45

a	-17	-5	3	6	12	21	142	45

Example (concl)

a	-17	-5	3	6	12	21	142	45

a	-17	-5	3	6	12	21	45	142

Sorting Analysis

- How many steps are needed to sort n things?
- For each swap, we have to search the remaining array
  - length is proportional to original array length n
- Need about n search/swap passes
- Total number of steps proportional to n^2
- Conclusion: selection sort is pretty expensive (slow) for large n

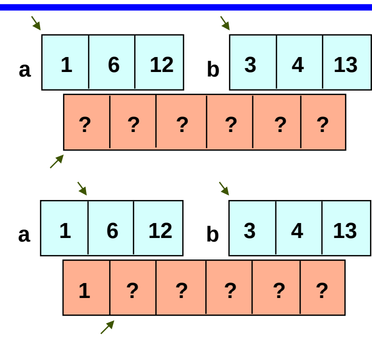
Can We Do Better Than n^2?

- ***Sure we can!***
- Selection, insertion, bubble sorts are all proportional to n^2
- Other sorts are proportional to $n \log n$
  - **Mergesort**
  - Quicksort
  - etc.
- As the size of our problem grows, the time to run a n^2 sort will grow much faster than a $n \log n$ one.

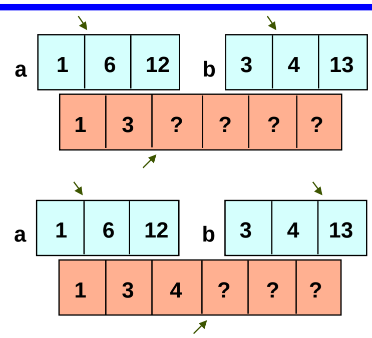
“Mergesort”

- We’ll see how to write this later, but for now we’ll see no C.
- **Basic idea:**
  - Start with some small sorted pieces: “runs”
  - Merge pairs of runs together to make larger sorted runs
  - When we finish merging the final pair, then we have sorted our array.
- **Basic operation is the merge.**

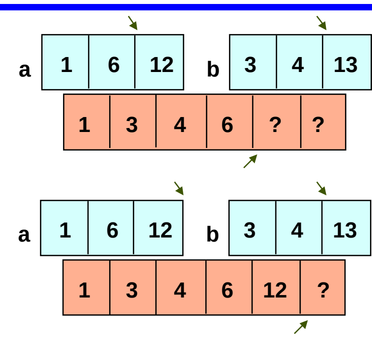
Subproblem: Merge



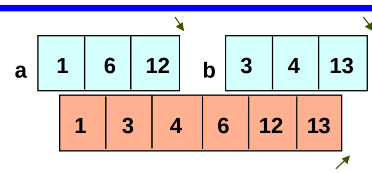
Subproblem: Merge



Subproblem: Merge



Subproblem: Merge



- We only used *n* comparisons and *n* copies so the amount of work we did was proportional to *n*.
- This is not a sorting algorithm yet!
  - How did we get the small runs to begin with?

Turning Merge into a Sort

- We need to have runs to merge them. Where do we find them?
- Answer: Individual elements are just little runs.
- **Mergesort:**
  - Merge runs of length 1 into runs of length 2
  - Merge the new runs of length 2 into runs of length 4
  - Merge the new runs of length 4 into runs of length 8
  - Continue until done

Example

a	3	12	-5	6	142	21	-17	45

Merge into runs of 2

a	3	12	-5	6	21	142	-17	45

Merge into runs of 4

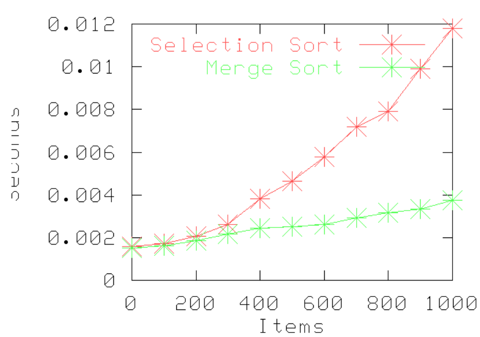
a	-5	3	6	12	-17	21	45	142

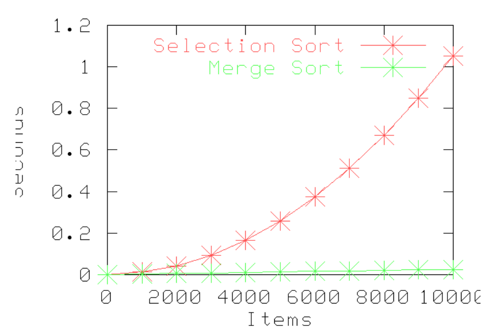
Example

Merge into run of 8

a	-17	-5	3	6	12	21	45	142

- Of course, now we're done.
- Each merge step took time proportional to *n*.
- How many merges steps did we use?
  - In this case 3.
  - In general we use $log_2 n$ merge steps because we double the size of our runs during each merge step.
- Total time is $n log_2 n$. (Or just *n log n*)





Any better than *n log n*?

- In general, no.
- In special cases, we can do even better:
  - Example: Sort exams by score: drop each exam in one of 101 piles; work is proportional to *n*
- **Curious fact:** efficiency can be studied and predicted mathematically, without using a computer at all!
- This branch of mathematics is called *complexity theory* and has many interesting, unsolved problems.

Comments about Efficiency

- Efficiency means doing things in a way that saves resources
  - Usually measured by *time* or *memory* used
- Many small programming details have little or no measurable effect on efficiency
- The big differences comes with the right choice of *algorithm* and/or *data structure*

L