ESC101: Introduction to Computing

Sorting

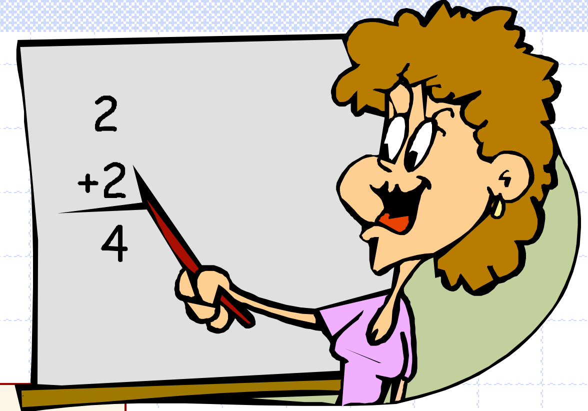
Sorting

- Given a list of integers (in an array), arrange them in ascending order.
  - Or descending order

INPUT ARRAY	5	6	2	3	1	4
OUTPUT ARRAY	1	2	3	4	5	6

- Sorting is an extremely important problem in computer science.
  - A common problem in everyday life.
  - Example:
    - Contact list on your phone.
    - Ordering marks before assignment of grades.

What's easy to do in a Sorted Array?



Clearly, searching for a key is fast.

Rank Queries: find the k^{th} largest/smallest value.
Quantile: 90%ile—the key value in the array such that 10% of the numbers are larger than it.

40	50	55	60	70	75	80	85	90	92

Marks in an exam: sorted

90 percentile : 90
80 percentile : 85
10 percentile: 40
50 percentile: 70
(also called median)

Sorted array have difficulty with

- inserting a new element while preserving the sorted structure.
- deleting an existing element (while preserving the sorted structure.
- In both cases, there may be need to shift elements to the right or left of the index corresponding to insertion or deletion.

40	50	55	60	70	75	80	85	90	92

Example: Insert 65.
1. Find index where 65 needs to be inserted

40	50	55	60	65	70	75	80	85	90	92

2. Shift right from index 5 to create space.

3. Insert 65

May have to shift n-1 elements in the worst case.

Sorting

- Many well known sorting Algorithms
  - Selection sort
  - Quick sort
  - Merge sort
  - Bubble sort
  - …
- Special cases also exist for specific problems/data sets
- Different runtime
- Different memory requirements

Selection Sort

- Select the largest element in your array and swap it with the first element of the array.
- Consider the sub-array from the second element to the last, as your current array and repeat Step 1.
- Stop when the array has only one element.
  - Base case, trivially sorted

Selection Sort: Pseudo code

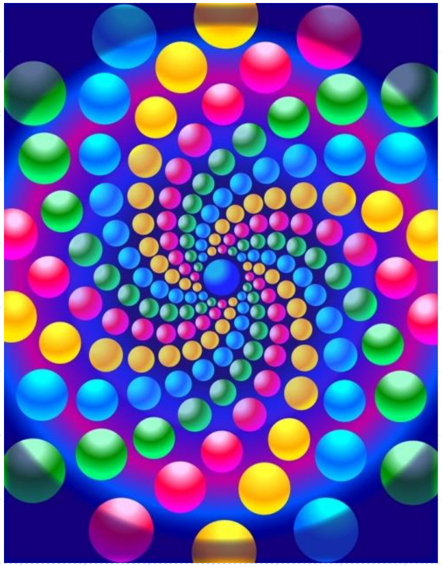
```
selection_sort(a, start, end)  {
    if (start == end) /* base case, one elt => sorted */
        return;

    idx_max = find_idx_of_max_elt(a, start, end);
    swap(a, idx_max, start);
    selection_sort(a, start+1, end);
}
```

```
swap(a, i, j)  {
    tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

```
main() {
    arr[] = { 5, 6, 2, 3, 1, 4 };
    selection_sort(arr, 0, 5);
    /* print arr */
}
```

Selection Sort: Properties



- Is the pseudo code iterative or recursive?
- What is the estimated run time when input array has **n** elements
  - for swap **Constant**
  - for find_idx_of_max_elt **$\propto$ n**
  - for selection_sort **On next slide**
- **Practice**: Write C code for iterative and recursive versions of selection sort.

Selection Sort: Time Estimate

- Recurrence

$$T(n) = T(n-1) + k_1 \times n + k_2$$

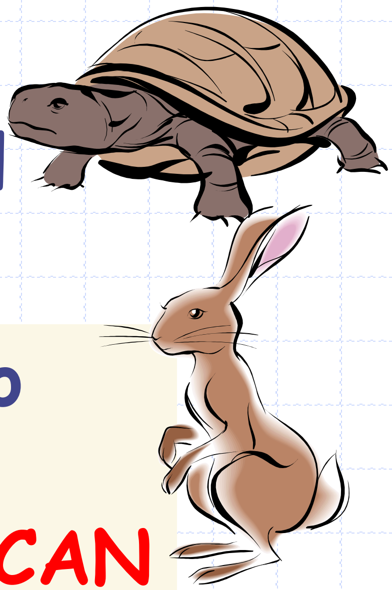
- Solution

$$T(n) \propto n(n+1)$$

- Or simply

$$T(n) \propto n^2$$

Selection sort runs in time proportional to the square of the size of the array to be sorted.

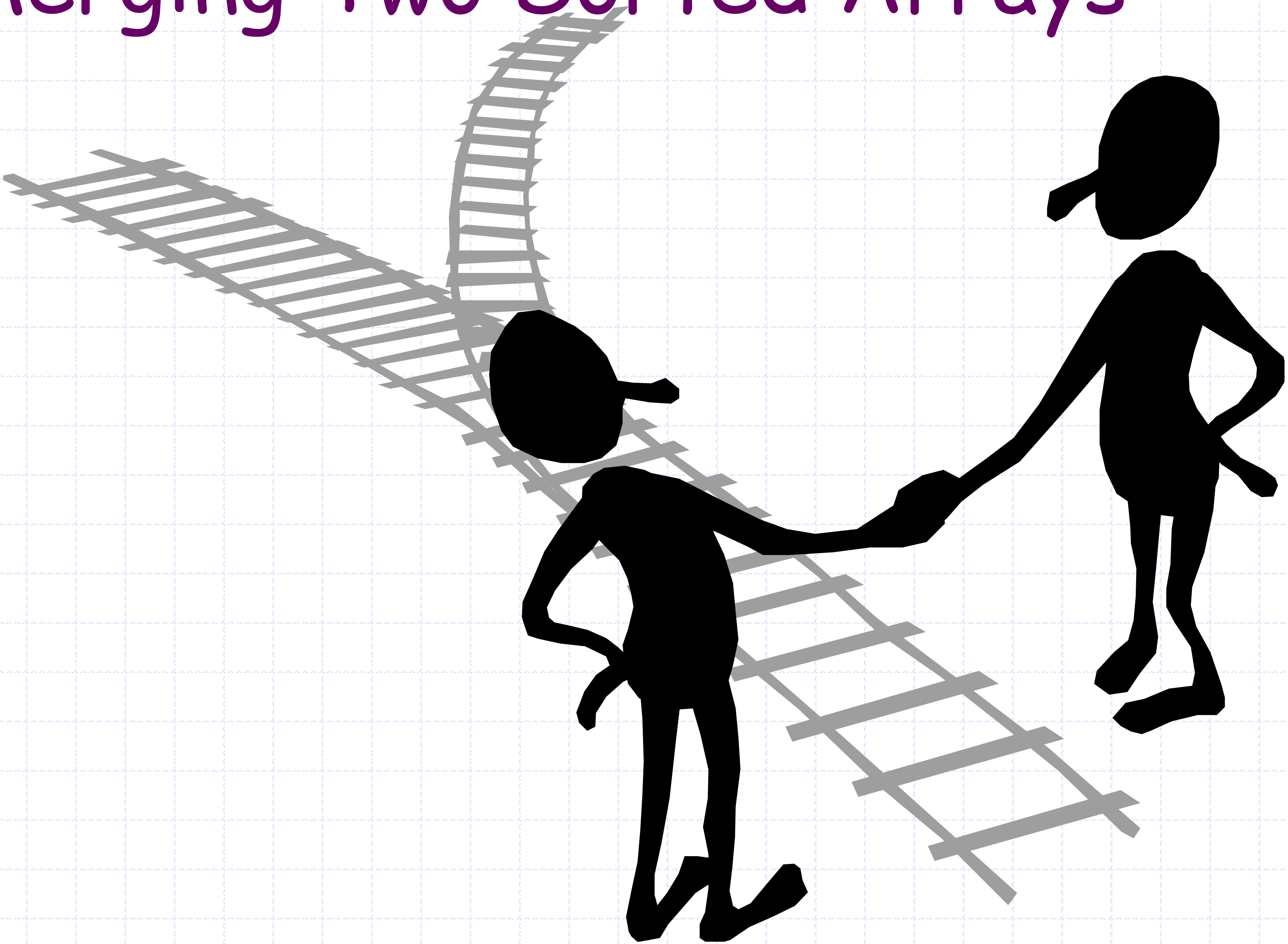


Can we do better?
YES WE CAN

Merging Two Sorted Arrays

- Input: Array *A* of size n & array B of size m.
- Create an empty array *C* of size n + m.
- Variables i , j and k
  - array variables for the arrays *A*, B and *C* resp.
- At each iteration
  - compare the i^{th} element of *A* (say u) with the j^{th} element of B (say v)
  - if u is smaller, copy u to *C*; increment i and k,
  - otherwise, copy v to *C*; increment j and k,

Merging Two Sorted Arrays



Time Estimate

- Number of steps $\propto 3(n + m)$.
  - The constant 3 is not very important as it does not vary with different sized arrays.
- Now suppose A and B are halves of an array of size n (both have size n/2).
- Number of steps = 3n.

$$T(n) \propto n$$

MergeSort

- Merge function can be used to sort an array
  - recursively!
- Given an array C of size n to sort
  - Divide it into Arrays A and B of size n/2 each (approx.)
  - Sort A into A′ using MergeSort
  - Sort B into B′ using MergeSort
  - Merge A′ and B′ to give C′ ≡ C sorted
- Can we reduce #of extra arrays (A′, B′, C′)?

Recursive calls.
Base case?
n <= 1

```
/*Sort ar[start, …, start+n-1] in place */
void merge_sort(int ar[], int start, int n) {
   if (n>1) {
      int half = n/2;
      merge_sort(ar, start, half);
      merge_sort(ar, start+half, n-half);
      merge(ar, start, n);
   }
}
```

```
int main() {
   int arr[]={2,5,4,8,6,9,8,6,1,4,7};
   merge_sort(arr,0,11);
   /* print array */
   return 0;
}
```

```
void merge(int ar[], int start, int n) {
  int temp[MAX_SZ], k, i=start, j=start+n/2;
  int lim_i = start+n/2, lim_j = start+n;
  for(k=0; k<n; k++) {
    if ((i < lim_i) && (j < lim_j)) {// both active
      if (ar[i] <= ar[j]) { temp[k] = ar[i]; i++; }
      else { temp[k] = ar[j]; j++; }
    } else if (i == lim_i) // 1st half done
      temp[k] = ar[j]; j++; // copy 2nd half
    else // 2nd half done
      temp[k] = ar[i]; i++; // copy 1st half
  }
  for (k=0; k<n; k++)
    ar[start+k]=temp[k]; // in-place
}
```

Time Estimate

`void merge_sort(int a[], int s, int n) {`	$T(n)$
`  if (n>1) {`	c
`    int h = n/2;`	c
`    merge_sort(a, s, h);`	$T(n/2)$
`    merge_sort(a, s+h, n-h);`	$T(n-n/2) \approx T(n/2)$
`    merge(a, s, n);`	$\approx 4n$
`  }`	
`}`	

Time Estimate

$T(n) = 2T(n/2) + 4n$

$= 2(2T(n/4) + 4n/2) + 4n = 2^2T(n/4) + 8n$

$= 2^2(2T(n/8) + 4n/4) + 4n = 2^3T(n/8) + 12n$

$= \ldots$ // keep going for k steps

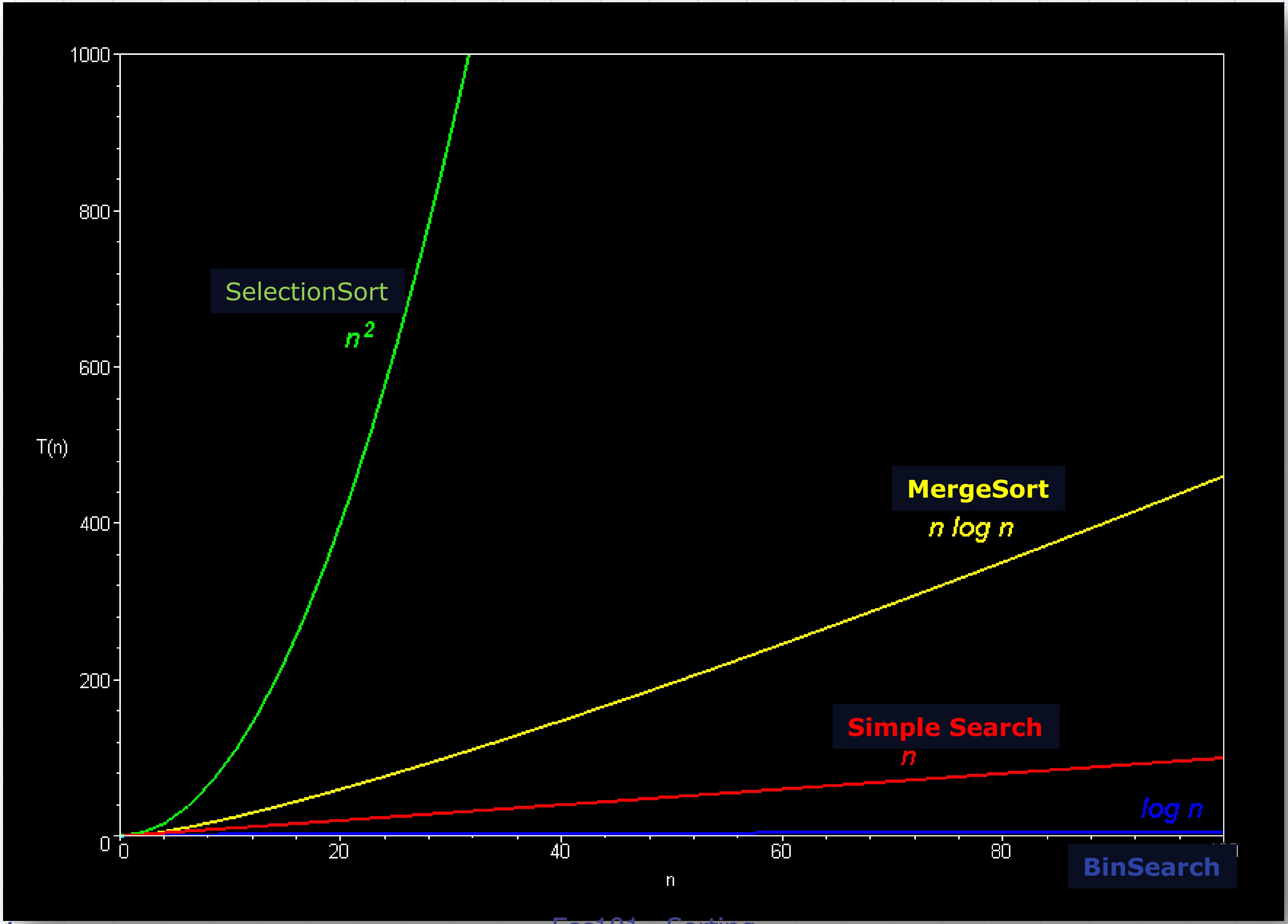
$= 2^kT(n/2^k) + k*4n$

Assume $n = 2^k$ for some k. Then,

$$T(n) = n*T(1) + 4n*\log_2 n$$

$$T(n) \propto n \log_2 n$$

Time Estimates…



Source: http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/