ESC101: Introduction to Computing

Sorting

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: Time Estimate

- Recurrence

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

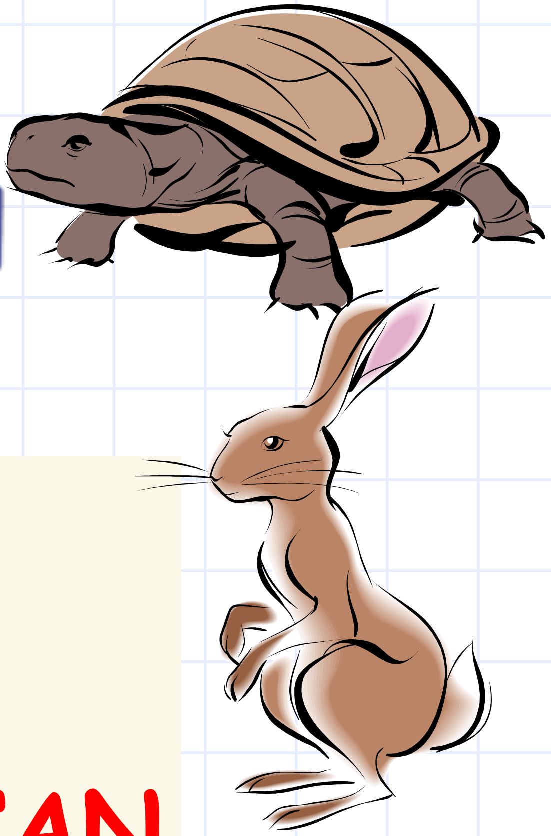
- Solution

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

- Or simply

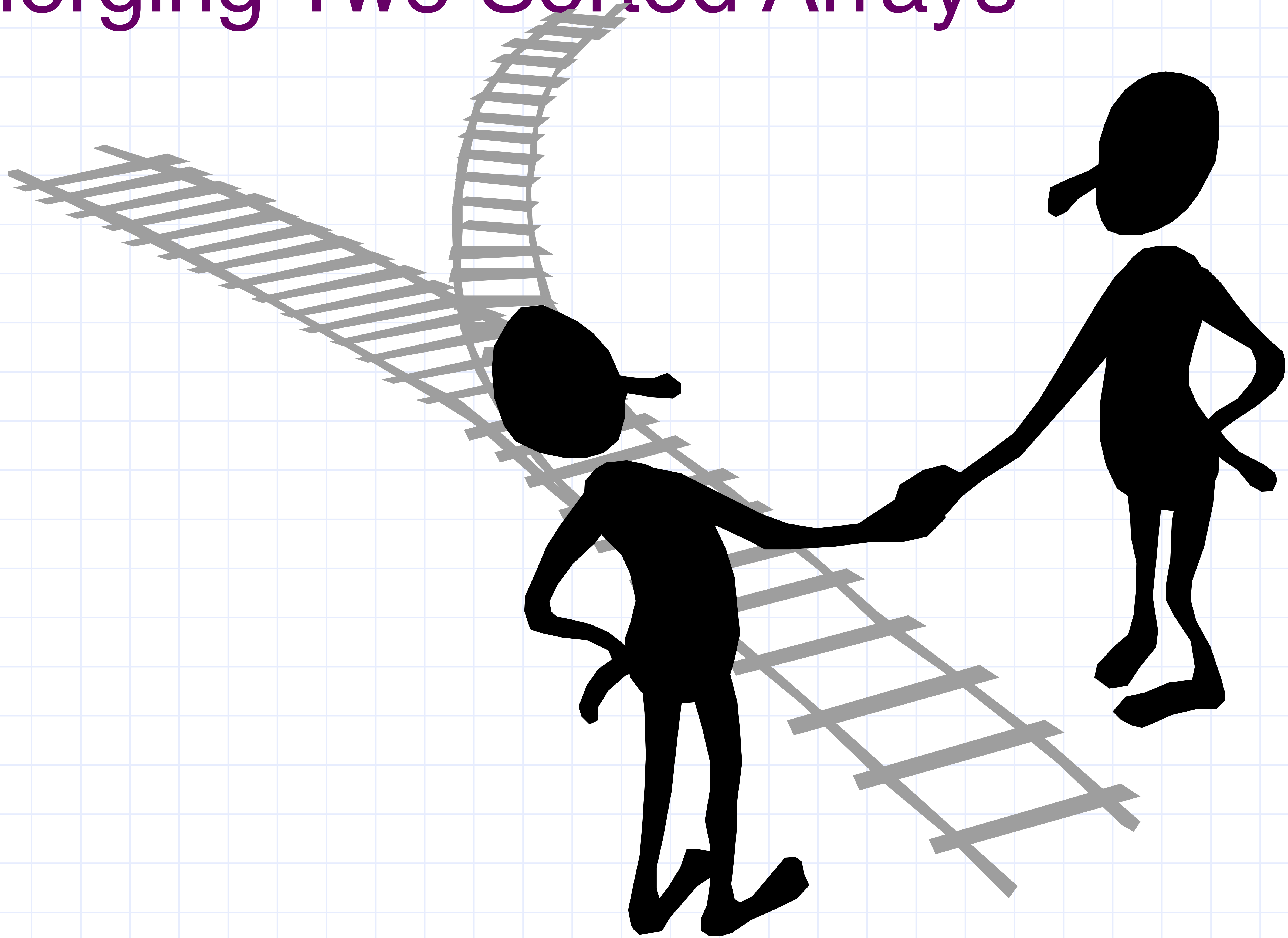
$$T(n) \propto n^2$$

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



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,

```c
#include <stdio.h>
int merge(int a[], int n1, int b[], int n2, int c[], int n3);

int main()
{
        int a[]={10,20, 30};
        int b[]={5,15, 25};
        int c[10];
        int n;
        n = merge(a, 2, b, 2, c, 0);
        for(int i=0; i<n; i++)
                printf("%d ",c[i]);
        printf("\n");
        return 0;
}
```

```c
int merge(int a[], int n1, int b[], int n2, int c[], int n3)
{
        if (n1 < 0 && n2 < 0)
                return n3;

        if(n1 < 0)
        {
                c[n3] = b[n2];
                n3++; n2--;
                return merge(a, n1, b, n2, c, n3); //could be done with a for loop too
        }

        if(n2 < 0)
        {
                c[n3] = a[n1];
                n3++; n1--;
                return merge(a, n1, b, n2, c, n3); //could be done with a for loop too
        }

        if(a[n1] > b[n2])
        {
                c[n3] = a[n1]; n3++; n1--;
        }
        else
        {
                c[n3] = b[n2]; n3++; n2--;
        }

        return merge(a, n1, b, n2, c, n3);

}
```

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.

$$T(n) \propto n$$

Merge Sort

- 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

```c
/*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);
  }
}
```

```c
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) { //n is no of elems
  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) {
   if (n>1) {
      int h = n/2;
      merge_sort(a, s, h);
      merge_sort(a, s+h, n-h);
      merge(a, s, n);
   }
}
```

$T(n)$

c

c

$T(n/2)$

$T(n-n/2) \approx T(n/2)$

$\approx 4n$

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

Time Estimate

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

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

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

= … // keep going for k steps

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

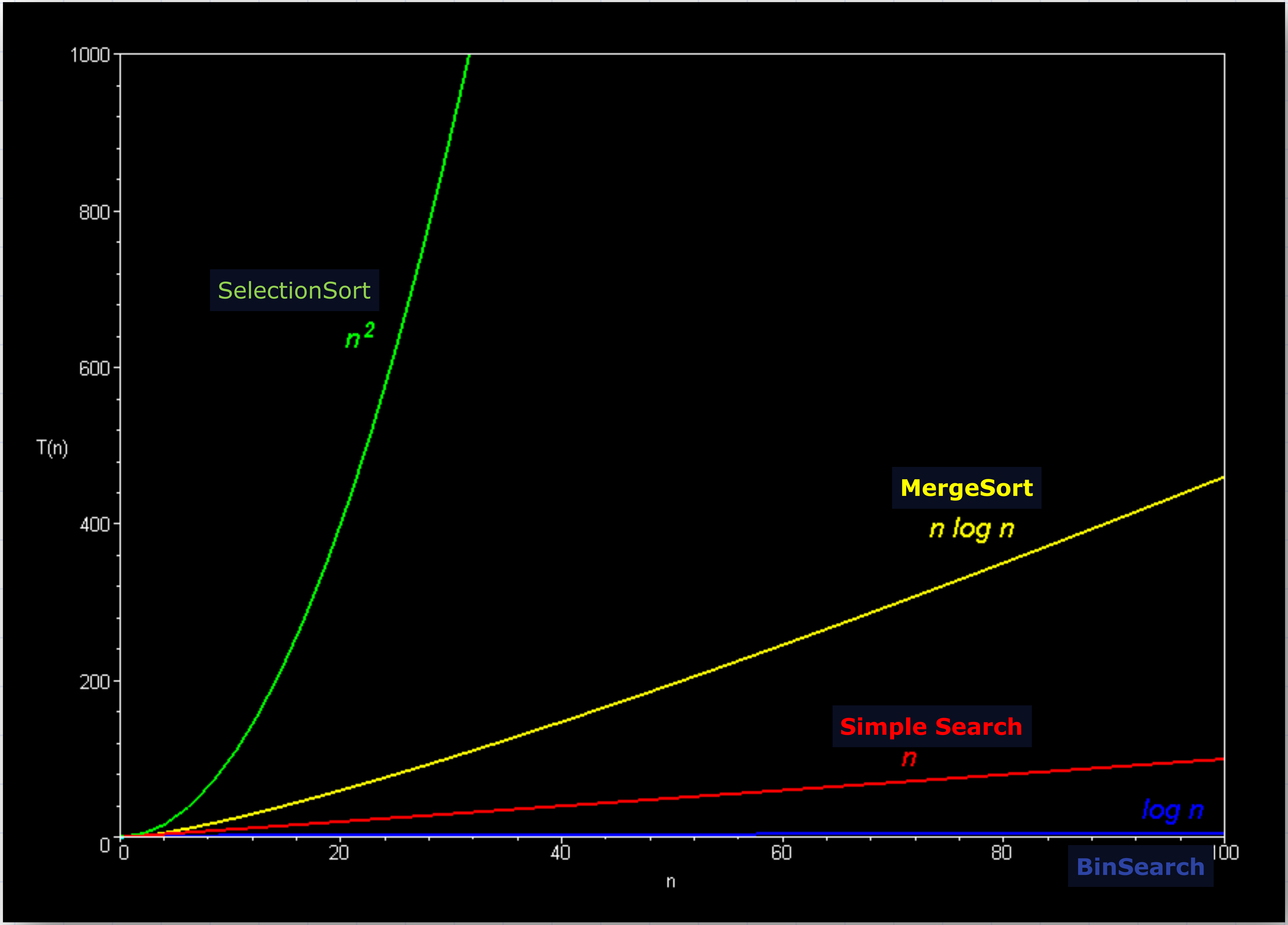
Assume $n = 2^k$ for some k. Then,

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

Time Estimates…

Why worry about O(n) vs O(n^2) vs O(…) algorithm?

Time Estimates…



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

Around Easter 1961, a course on ALGOL 60 was offered …
It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining.

It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who **included recursion in their language** and enabled me to describe my invention so elegantly to the world.

I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed.

- **C. A. R. Hoare, ACM Turing Award Lecture, 1980**

QuickSort - Partition Routine

A useful sub-routine (function) for many problems, including quicksort, one of the popular sorting methods.

1. **Partition takes an array a[] of size n and a value called the pivot.**
2. **The pivot is an element in the array, for instance, a[0].**
3. **Partition re-arranges the array elements into two parts:**
   a) **the left part has all elements <= pivot.**
   b) **the right part has all elements >= pivot.**
4. **Partition returns the index of the beginning of the right part.**

Let us see an example.

1. **Partition** takes an array a[] of size n and a value called the pivot.
2. The pivot is an element in the array, for instance, a[0].
3. Partition re-arranges the array elements into two parts:
   a) all elements in the left part are <= pivot
   b) all elements in the right part are >= pivot

Input Array a[], size is n : 11

31	4	10	35	59	31	3	25	35	11	0

Pivot element is assumed to be a[0]: 31

0	4	10	11	25	3	31	59	35	35	31

left partition

right partition

Observations

Multiple “partitions” of an array are possible, even for the same pivot. They all would satisfy the above specification.

Note: Partition **DOES NOT** sort the array. It is “weaker” than sorting. But it is useful step towards sorting (useful for other problems also).