Running Time and Sorting

Running Time

- How fast does an algorithm run?
- Usually relative to the size of the input
- Is it reasonable? T(n), T(log n), T(n^2)...
- For even moderate input sizes is it unreasonable? T(2^n)

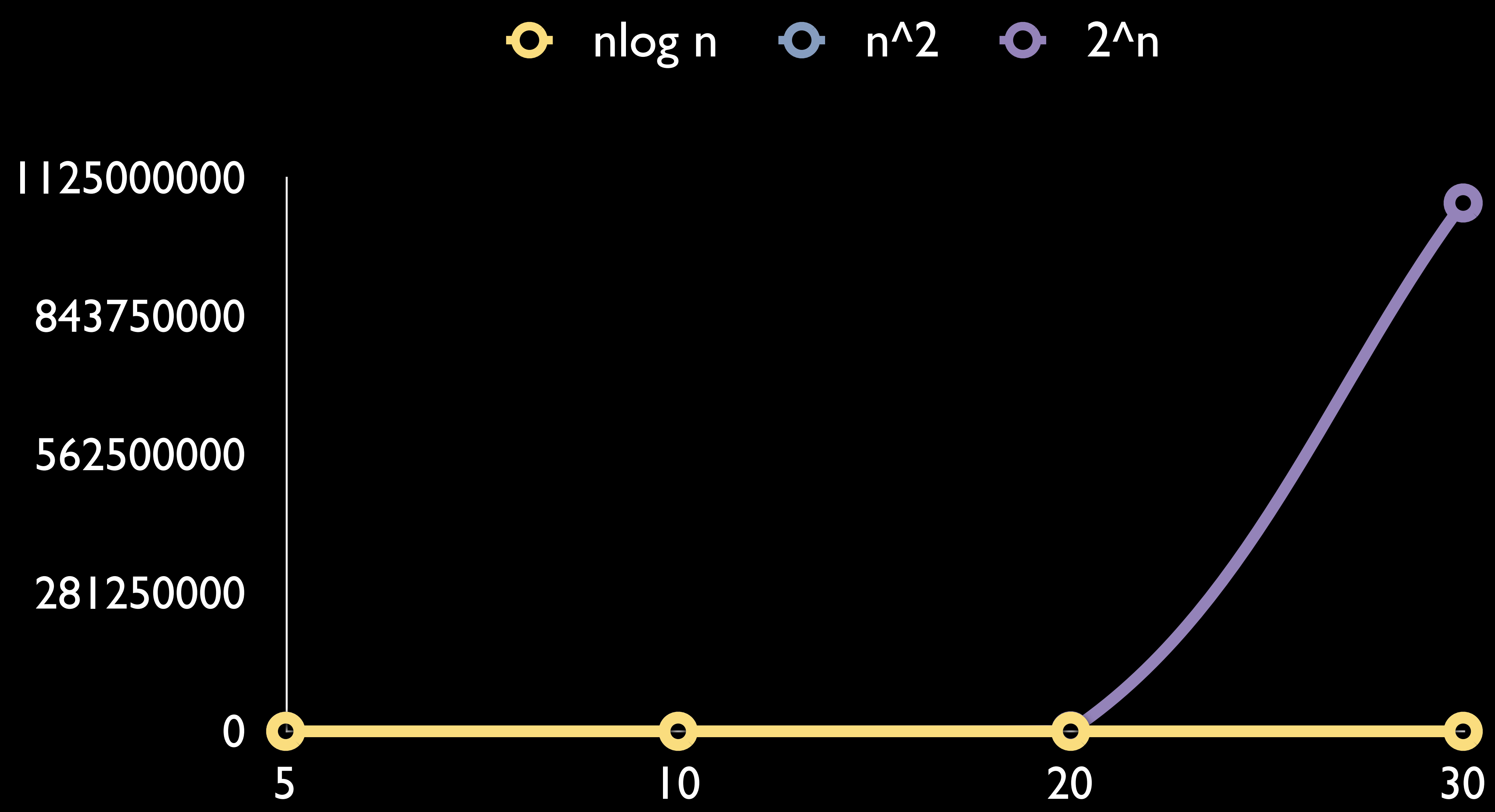
Examples of running times (in milli seconds)

input size	*nlogn*	n^2	2^n
1	0	1	2
10	10	100	1024
100	200	10000	1.26765E+30
1000	3000	1000000	1.07151E+301

3.9x10^19 years!

2.4 x 10^290 years!

Function growth



Computing Running Time

- Which parts depend on input size?
- Usually ignore things that aren't input size dependent: constant amount of work on any input
- Places where time is often needed:
  - loops
  - recursive calls

Computing Running Time: Loops

Consider the following two loops:

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

Which one takes less time?

A

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

B

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

What are their rough running times?

A

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

B

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

Computing Running Time: Loops

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

n iterations

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

Computing Running Time: Loops

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
```

n iterations

```
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

n iterations

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

Computing Running Time: Loops

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

n iterations

n iterations

Total: n+n=2n

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

Computing Running Time: Loops

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

n iterations

n iterations

Total: 2n

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

n iterations

Computing Running Time: Loops

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

n iterations

n iterations

Total: 2n

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

n iterations

n iterations in each of n iterations

Computing Running Time: Loops

```
for (int i = 0; i< n; i++)
{
    if list[i] > value
        remove list[i] from list
}
for (int j = 0; j<n; j++)
{
    display(list[j])
}
```

n iterations

n iterations

Total: 2n

```
for (int i = 0; i<n; i++)
{
    for(int j=0; j<n; j++)
    {
        display(i*j);
    }
}
```

n iterations

n iterations
in each of n iterations

Total: n*n=n^2

Recursive Calls: Quicksort

```
Quicksort(list a, length x)
if(x<=1) return a;
int p = x-1;
list lower;
list higher;
higher.add(p)
for(int i =0; i<x-1, i++)
      if(a[i] < p)
            lower.add(a[i]);
      else
            higher.add(a[i]);
Quicksort(lower, lower.length());
Quicksort(higher, higher.length());
return(lower+higher);
```

Recursive Calls: Quicksort

```
QUICKSORT(list a, length x)
      if(x<=1) return a;
      int p = x-1;
      list lower;
      list higher;
      higher.add(p)
      for(int i =0; i<x-1, i++)
          if(a[i] < p)
              lower.add(a[i]);
          else
              higher.add(a[i]);
      QUICKSORT(lower, lower.length());
      QUICKSORT(higher, higher.length());
      return(lower+higher);
```

what's going on?

Running time of Quicksort

- Each time the pivot splits the list into two lists
- At each call there at most ~ n^2 comparisons
- consider how it works for: 1 2 3 4 5
- how about: 1 2 4 5 3

Example: quicksort

1 2 3 4 5

Running time of Quicksort

- Each time the pivot splits the list into two lists
- At each call there at most ~ n^2 comparisons
- consider how it works for: 1 2 3 4 5
- how about: 1 2 4 5 3

~n^2 comparisons

Example: quicksort

1 2 4 5 3

Running time of Quicksort

- Each time the pivot splits the list into two lists
- At each call there at most ~ n^2 comparisons
- consider how it works for: 1 2 3 4 5
- how about: 1 2 4 5 3

~n^2 comparisons

~n log n comparisons

Which Sort is best?

- Many things to consider:
  - range of input
  - type of input
    - unique values or not?
    - integer, real numbers, alphabetical...
  - expected size of input

Can we sort better than T(n log n)?

- What if we know something about the data?
- Sorting integers: create a bucket for ranges of values
- if needed, use a comparison base sort in each bucket

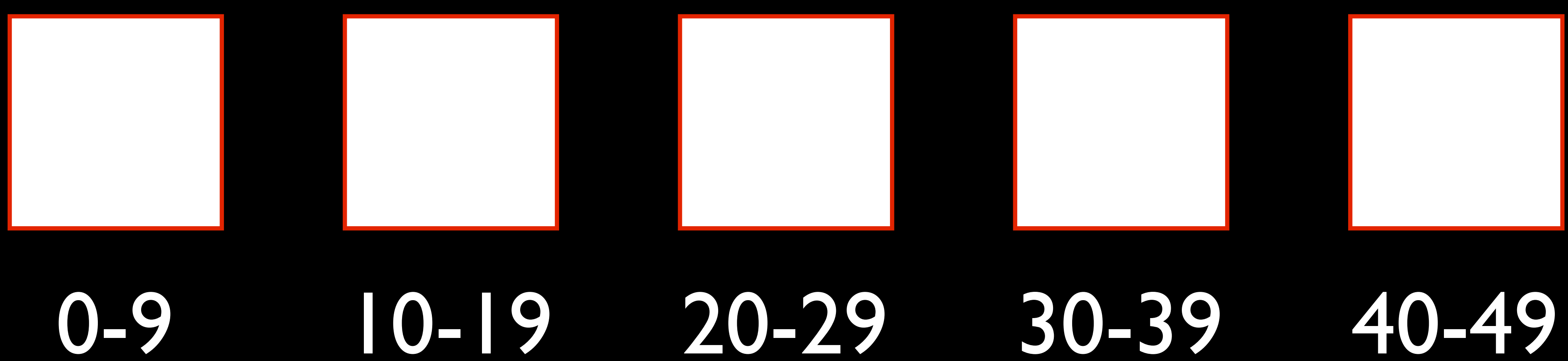
Bucket Sort Example

input: 20, 33, 2, 42, 48, 10, 17, 35, 9

Bucket Sort Example

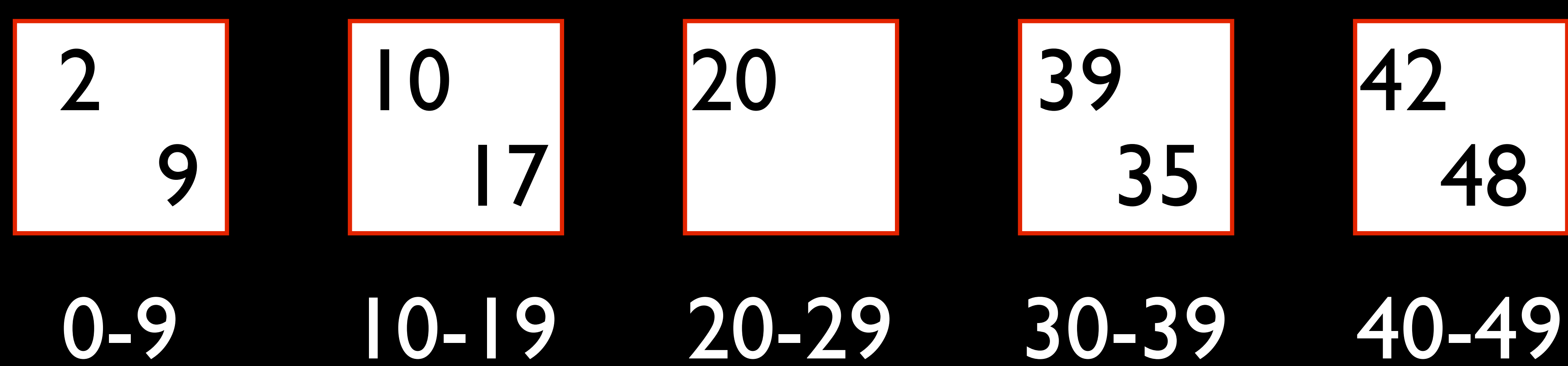
If we know the input is always integers from 0 to 49

input: 20, 33, 2, 42, 48, 10, 17, 35, 9



Bucket Sort Example

input: 20, 39, 2, 42, 48, 10, 17, 35, 9



Bucket Sort Example

input: 20, 39, 2, 42, 48, 10, 17, 35, 9

One pass: n comparisons

0-9	10-19	20-29	30-39	40-49
2	10	20	39	42
9	17		35	48

Some Sorting Algorithms ...

- Selection sort
- Insertion sort
- Quicksort
- Bubble sort
- Heap sort
- Bucket sort
- Merge sort
- and many more...

Properties of sorting algorithms

- comparison based?
- speed (n logn vs. n^2)
- memory usage (in place or not?)
- iterative or recursive?
- stability (do like-valued items stay in original order?)

Properties of some sorting algorithms

- Selection sort: in place, comparison, slow (~n^2)
- Insertion sort: low memory needs, comparison, shifting needed (~n^2)
- Quicksort: not stable, fast (~nlogn), low memory
- Bubble sort: simple, slow, low memory (~n^2)
- Bucket sort: fast for data with little variation (~n)
- Merge sort: fast (~nlogn)