CSE 2320

Week of 06/22/2020

Instructor : Donna French

OLQ2

```c
#include <stdio.h>

int FunctionR(int z)
{
    static int y = 0;

    if(z != 0)
    {
        y++;
        FunctionR(z/10);
    }

    return y;
}

int main(void)
{
    int x = 12335;

    printf("%d", FunctionR(x));

    return 0;
}
```

main() call FunctionR with 12335

FunctionR – Pass 1
z = 12335
y = 0
z != 0 so y = 1
Call FunctionR with 1233
return y

FunctionR – Pass 2
z = 1233
y = 1
z != 0 so y = 2
Call FunctionR with 123
return y

FunctionR – Pass 3
z = 123
y = 2
z != 0 so y = 3
Call FunctionR with 12
return y

FunctionR – Pass 4
z = 12
y = 3
z != 0 so y = 4
Call FunctionR with 1
return y

FunctionR – Pass 5
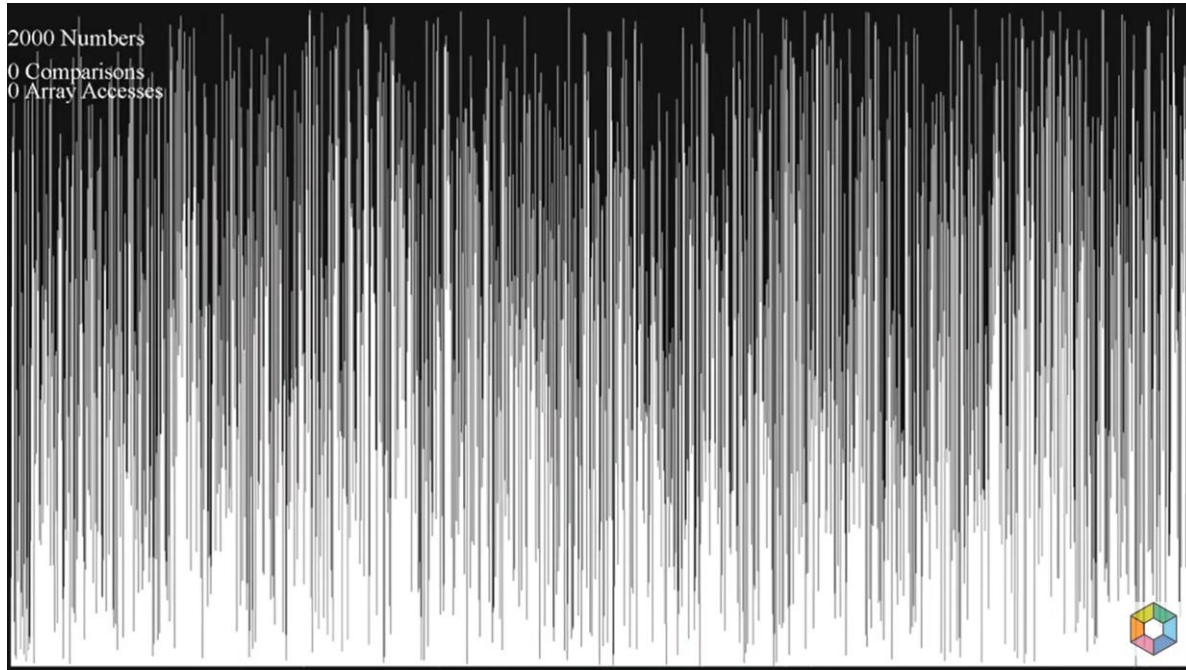z = 1
y = 4
z != 0 so y = 5
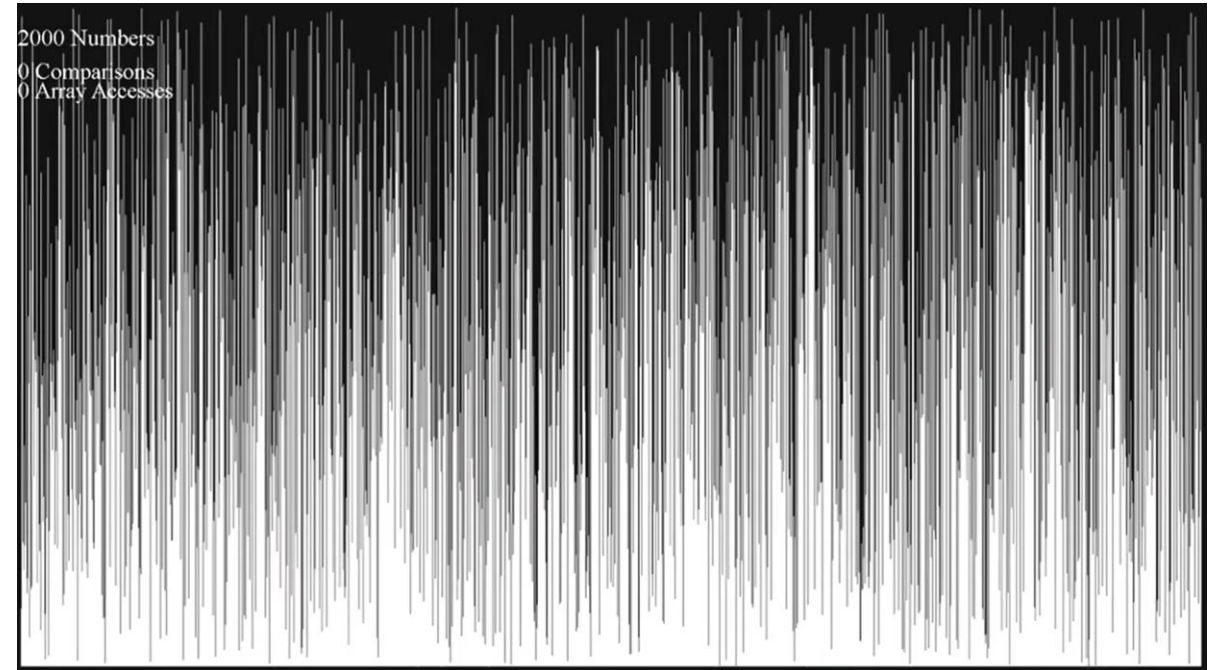Call FunctionR with 0
return y

FunctionR – Pass 6
z = 0
return y (5)

The Role of Algorithms in Computing

Insertion Sort n^2

Merge Sort $n \log_2 n$

Designing Algorithms

Insertion sort uses an incremental approach

having sorted the subarray $A[1..j-1]$, we inserted the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1..j]$

We are going to examine an alternative design approach called

divide and conquer

Designing Algorithms

Many useful algorithms are recursive in structure

To solve a given problem, an algorithm calls itself recursively one or more times to deal with closely related subproblems.

These types of algorithms follow a divide and conquer approach

They break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively and the combine these solutions to create a solution to the original problem.

The Divide and Conquer Approach

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblems are small enough, just solve them by brute force.

Combine the subproblem solutions to give a solution to the original problem

Merge Sort

Merge Sort is a sorting algorithm based on divide and conquer.

Its worst-case running time has a lower order of growth than insertion sort.

Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p \ldots r]$

Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems

Merge Sort

To sort `A[p .. r]`

Divide by splitting into two subarrays `A[p .. q]` and `A[q+1 .. r]` where q is the halfway point of `A[p .. r]`

Conquer by recursively sorting the two subarrays `A[p .. q]` and `A[q+1 .. r]`

Combine by merging the two sorted subarrays `A[p .. q]` and `A[q+1 .. r]` to produce a single sorted subarray `A[p .. r]`. To accomplish this step, we'll define a procedure `Merge(A, p, q, r)`

The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted – can't divide 1 value.

Merge Sort

MergeSort(*A, p, r*)

    if *p < r*   // check for base case

       $q = (p + r)/2$ // divide

       MergeSort (*A, p, q*)   // conquer
       MergeSort(*A, q+1, r*)   // conquer

       Merge (*A, p, q, r*)   // combine

```c
int main()
{
    int arr[] = {12, 11, 5, 13, 7, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);

    return 0;
}
```

```c
void mergeSort(int arr[], int L, int R)
{
    if (L < R)
    {
        int M = (L+R)/2;

        mergeSort(arr, L, M);

        mergeSort(arr, M+1, R);

        merge(arr, L, M, R);
    }
}
```

{12, 11, 5, 13, 7, 6};

M = (L + R)/2

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

MS MS	
L =	R =
if L < R, then M =	
MS -> L, M	MS -> M+1, R
merge	

```c
int main(void)
{
    int arr[] = {12, 11, 5, 13, 7, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);

    return 0;
}
```

```c
void mergeSort(int arr[], int L, int R)
{
    if (L < R)
    {
        int M = (L+R)/2;

        mergeSort(arr, L, M);

        mergeSort(arr, M+1, R);

        merge(arr, L, M, R);
    }
}
```

A-merge with L=0 M=0 R=1

B-merge with L=0 M=1 R=2   D-merge with L=3 M=4 R=5

C-merge with L=3 M=3 R=4   E-merge with L=0 M=2 R=5

$$\text{MERGE}(A, p, q, r)$$

```
n₁ = q − p + 1
n₂ = r − q
let L[1 .. n₁ + 1] and R[1 .. n₂ + 1] be new arrays
for i = 1 to n₁
        L[i] = A[p + i − 1]
for j = 1 to n₂
        R[j] = A[q + j]
L[n₁ + 1] = ∞
R[n₂ + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else A[k] = R[j]
                j = j + 1
```

p = 1, q = 7, r = 13

n₁ = q − p + 1 = 7−1+1 = 7
n₂ = r − q = 13 − 7 = 6

L[1...n₁ + 1] = L[1...8]
R[1...n₂ + 1] = R[1...7]

Merge – Part 1

```c
void merge(int arr[], int left, int middle, int right)
{
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
```

Merge – Part 2

```c
    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

Merge – Part 3

```c
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }


    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Merge – Part 1

```
void merge(int arr[], int left, int middle, int right)
{
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
```

A-merge with
left=0 middle=0 right=1

{12, 11, 5, 13, 7, 6}

L = {12} R = {11}

n1 = 1   n2 = 1

```
i = 0;
j = 0;
k = left;
```

Merge – Part 2

```
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

A-merge with
left=0 middle=0 right=1
L = {12} R = {11} n1=1 n2=1

```
{12, 11, 5, 13, 7, 6}
{11, 12, 5, 13, 7, 6}
```

```
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
```

Merge – Part 3

```
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}
```

Merge – Part 1

```
void merge(int arr[], int left, int middle, int right)
{
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
```

B merge with
left=0 middle=1 right=2
```
{11, 12, 5, 13, 7, 6}
```

L = {11,12} R = {5}

n1 = 2   n2 = 1

```
i = 0;
j = 0;                  Merge – Part 2
k = left;

while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

B-merge with
left=0 middle=1 right=2
L = {11,12} R = {5} n1=2 n2=1

```
{11, 12, 5, 13, 7, 6}
{5, 11, 12, 13, 7, 6}
while (i < n1)
{
        arr[k] = L[i];
        i++;
        k++;
}
                        Merge – Part 3

while (j < n2)
{
        arr[k] = R[j];
        j++;
        k++;
}
```

Merge – Part 1

```
void merge(int arr[], int left, int middle, int right)
{
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
```

C-merge with
left=3 middle=3 right=4

{5, 11, 12, 13, 7, 6}

L = {13} R = {7}

n1 = 1   n2 = 1

```
i = 0;
j = 0;              Merge – Part 2
k = left;

while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

C-merge with
left=3 middle=3 right=4
L = {13} R = {7} n1=1 n2=1

```
{5,  11,  12,  13,  7,  6}
{5,  11,  12,  7,  13,  6}
```

```
while (i < n1)
{
        arr[k] = L[i];
        i++;
        k++;
}
                        Merge – Part 3

while (j < n2)
{
        arr[k] = R[j];
        j++;
        k++;
}
```

Merge – Part 1

```
void merge(int arr[], int left, int middle, int right)
{
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
```

D-merge with
left=3 middle=4 right=5

```
{5, 11, 12, 7, 13, 6}
```

L = {7,13} R = {6}

n1 = 2   n2 = 1

```
i = 0;
j = 0;
k = left;
```

Merge – Part 2

```
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

D-merge with
left=3 middle=4 right=5
L = {7,13} R = {6} n1=2 n2=1

```
{5, 11, 12, 7, 13, 6}
{5, 11, 12, 6, 7, 13}
```

```
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
```

Merge – Part 3

```
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}
```

Merge – Part 1

```
void merge(int arr[], int left, int middle, int right)
{
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];
```

E-merge with
left=0 middle=2 right=5

{5, 11, 12, 6, 7, 13}

L = {5,11,12} R = {6,7,13}

n1 = 3   n2 = 3

```
i = 0;
j = 0;
k = left;
```

Merge – Part 2

```
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

E-merge with
left=0 middle=2 right=5
L = {5,11,12} R = {6,7,13} n1=3 n2=3

```
{5, 11, 12, 6, 7, 13}
{5, 6, 7, 11, 12, 13}
```

```
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
```

Merge – Part 3

```
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}
```

A merge {12} with {11}

$\{$ 12, 11, 5, 13, 7, 6 $\}$
$\{$ 11, 12, 5, 13, 7, 6 $\}$

B merge {11,12} with {5}

$\{$ 11, 12, 5, 13, 7, 6 $\}$
$\{$ 5, 11, 12, 13, 7, 6 $\}$

C merge {13} with {7}

$\{$ 5, 11, 12, 13, 7, 6 $\}$
$\{$ 5, 11, 12, 7, 13, 6 $\}$

D merge {7,13} with {6}

$\{$ 5, 11, 12, 7, 13, 6 $\}$
$\{$ 5, 11, 12, 6, 7, 13 $\}$

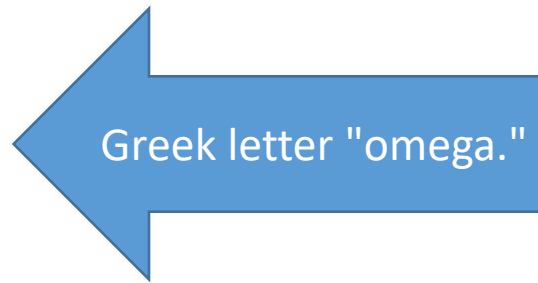E merge {5,11,12} with {6,7,13}

$\{$ 5, 11, 12, 6, 7, 13 $\}$
$\{$ 5, 6, 7, 11, 12, 13 $\}$

Asymptotic Notation

Sometimes, we want to say that an algorithm takes *at least* a certain amount of time, without providing an upper bound.
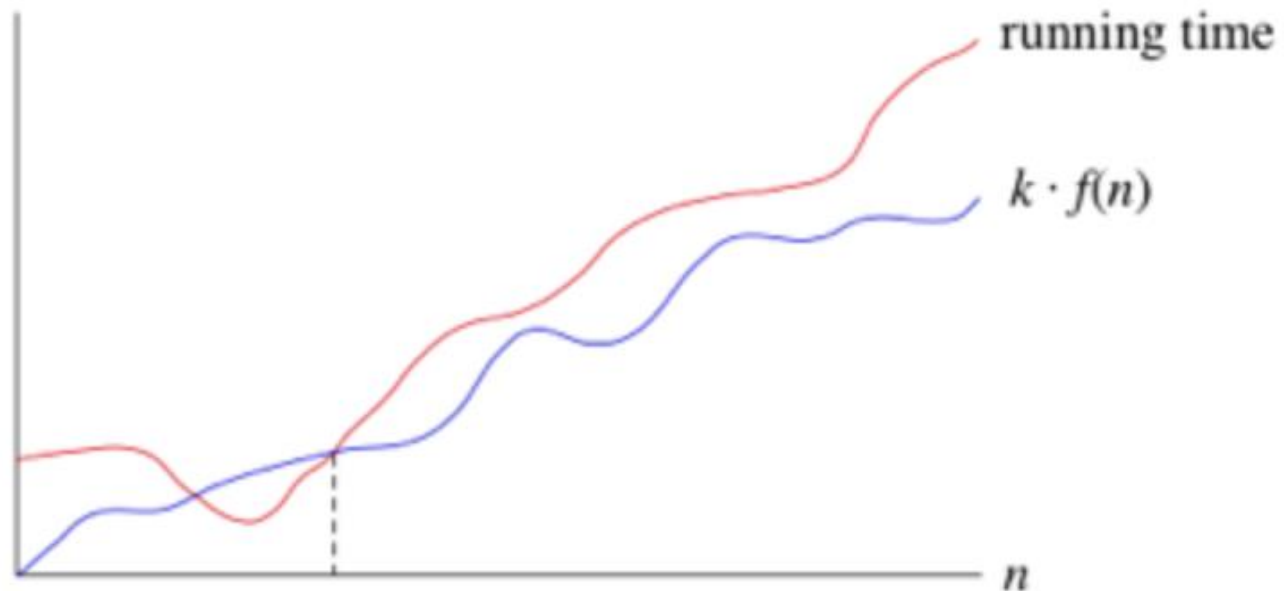
We use big-Ω notation

Greek letter "omega."

If a running time is $\Omega(f(n))$, then for large enough n, the running time is at least $k \cdot f(n)$ for some constant k.

Asymptotic Notation

Here's how to think of a running time that $\Omega(f(n)$

Asymptotic Notation

The running time is "big-Ω of $f(n)$"

Big-Ω notation is used for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

Just as $\Theta(f(n))$ automatically implies $O(f(n))$, it also automatically implies $\Omega(f(n))$.

So we can say that the worst-case running time of binary search is $\Omega(\log_2 n)$.

Asymptotic Notation

We can also make correct, but imprecise, statements using big-Ω notation.

For example, if you really do have a million dollars in your pocket, you can truthfully say

"I have an amount of money in my pocket, and it's **at least** 10 dollars."

That is correct, but certainly not very precise.

Similarly, we can correctly but imprecisely say that the worst-case running time of binary search $\Omega(1)$ because we know that it takes **at least** constant time.

Asymptotic Notation

$$f(n) = 5n^2 + 2n + 1$$

if f(n) is O(g(n))

    f(n) grows asymptotically no faster than g(n)
then

$$g(n) = n^2$$

if f(n) is Ω(g(n))

    f(n) grows asymptotically no slower than g(n)

if f(n) is Θ(g(n))

    f(n) grows asymptotically at the same rate as g(n)

Asymptotic Notation

Think of O as an upper bound and Ω as a lower bound.

These bounds can be tight or loose, but we prefer to make them tight as possible.

If we have tight bounds where O and Ω have the same growth rate, then we precisely know the growth rate.

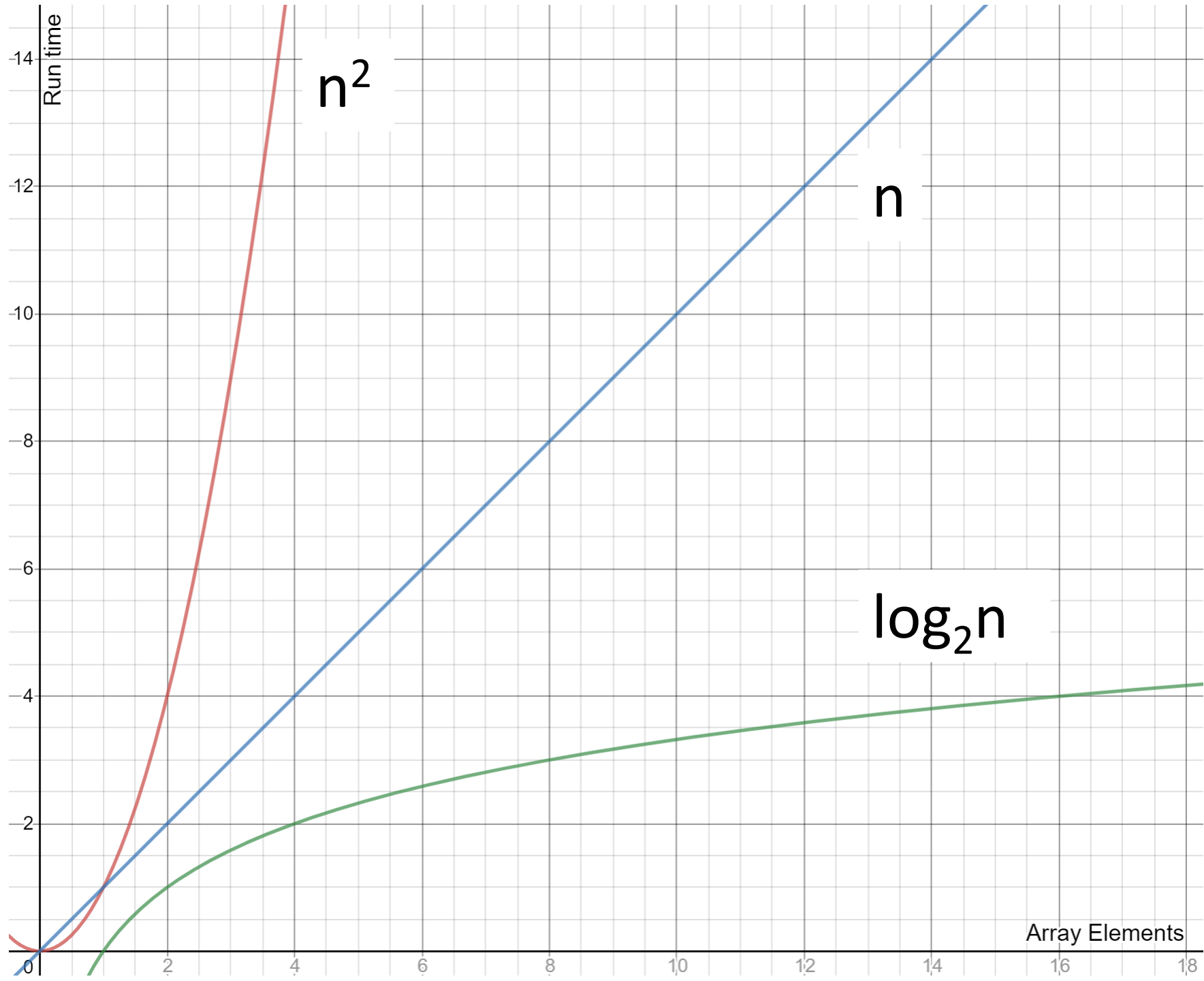If we can precisely give the growth rate, then we know Θ.

Asymptotic Notation

For $\log_2 n$…

$\log_2 n$ is $O(n^2)$ since $\log_2 n$ grows asymptotically no faster than n^2

$\log_2 n$ is also $O(n)$ since $\log_2 n$ grows asymptotically no faster than n

$\log_2 n$ is also $O(\log_2 n)$ since $\log_2 n$ grows asymptotically no faster than $\log_2 n$

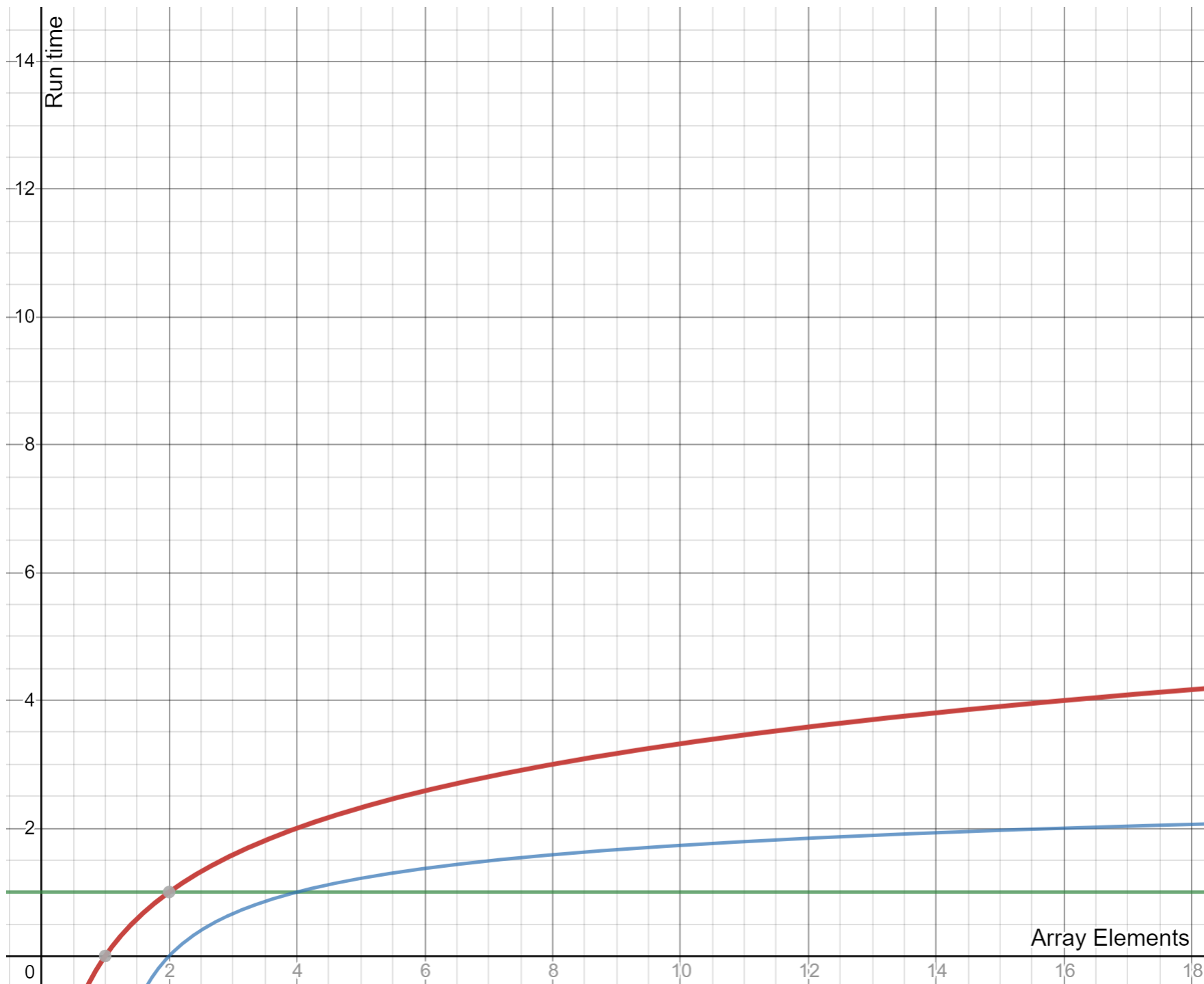We went from loose upper bounds to a tight upper bound

Asymptotic Notation

For $\log_2 n$…

$\log_2 n$ is $\Omega(1)$ since $\log_2 n$ grows asymptotically no slower than 1

$\log_2 n$ is $\Omega(\log_2(\log_2 n))$ since $\log_2 n$ grows asymptotically no slower than $\log_2(\log_2(n))$

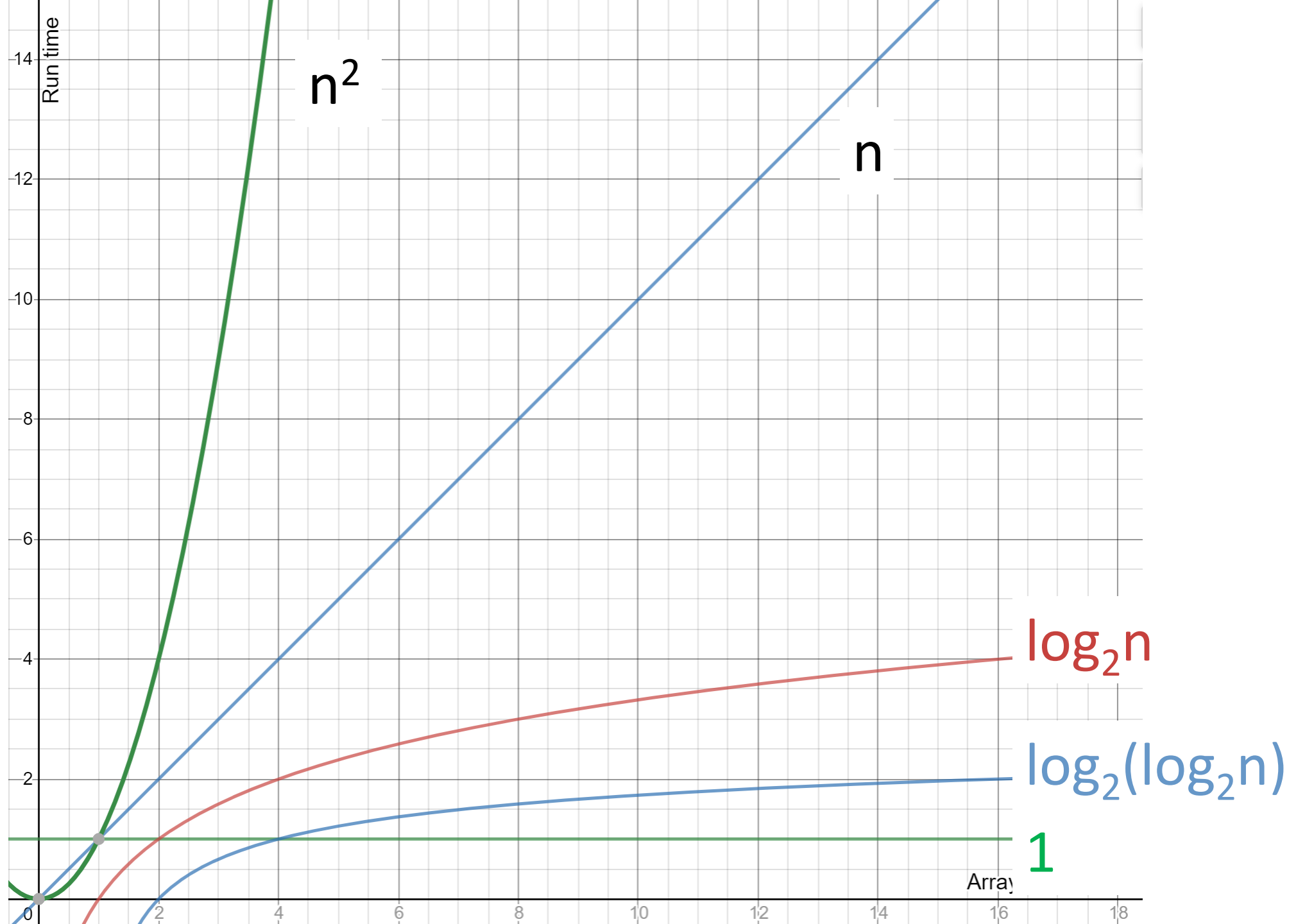$\log_2 n$ is $\Omega(\log_2 n)$ since $\log_2 n$ grows asymptotically no slower than $\log_2 n$

We went from loose lower bounds to a tight lower bound

Asymptotic Notation

Since we have $\log_2 n$ is

$O(\log_2 n)$

and

$\Omega(\log_2 n)$

we can say that $\log_2 n$ is $\Theta(\log_2 n)$

Asymptotic Notation

If we know the big Θ of a function, is there a reason that we would want to know a different big O or big Ω for the function?

It seems that if you have the big Θ, you already know the best big-O and big Ω.

Why would we want to know these other values? Are they significantly easier to find than the big Θ?

Does big Θ give the best big O and big Ω possible?

Asymptotic Notation

Typically, if we design an algorithm and perform some quick analysis on it, we will find a reasonable O value – meaning it doesn't have to be tight, but it shouldn't be too loose.

Often, that's all we care about.

Sometimes, we then can find a reasonable Ω value (again it doesn't have to be tight, but it shouldn't be too loose).

If it is critical to have tight bounds, or if the algorithm is easy to analyze, we can tighten up the bounds on O and Ω until they are the same and then we have Θ.

Asymptotic Notation

Let's say we have some task that our program needs to perform.

Often, we don't care about which algorithm we use as long as the asymptotic complexity of the algorithm is under some limit

the algorithm is fast enough for our needs.

If we write an algorithm that gets the job done, but it is hard to analyze, we can often make simplifying assumptions that make the analysis easier but the bounds looser.

Asymptotic Notation

```
for (i = 1; i <= n; i++)
{
    int number = result_of_a_very_hard_to_analyze_calculation;
    if number < 1
        number = 1;
    if number > 1
        number = n;
    for (k = 1; k <= number; k++)
        some_simple_stuff
}
```

Asymptotic Notation

```
for (i = 1; i <= n; i++)
{
    int number = result_of_a_very_hard_to_analyze_calculation;
    if number < 1
        number = 1;
    if number > 1
        number = n;
    for (k = 1; k <= number; k++)
        some_simple_stuff
}
```

outer for loop will loop n times

inner for loop will loop
   worst case = n times
   best case = 1 time

Running time is $O(n^2)$ and $\Omega(n)$

Since the calculation is hard to analyze, we might not be able to tighten up Ω or O.

If we can't tighten up the bounds on Ω or O, then we can't figure out Θ.

```c
#include <stdio.h>

void PrintArray(int A[], int n)
{
    int i = 0;
    for (i = 0; i < n; i++)
        printf("%d ", A[i]);

    printf("\n");
}

int X(int A[], int n, int x, int i)
{
    if (i == n)
        return 1;

    int z = A[i];

    int y = X(A, n, x * A[i], i + 1);

    A[i] = x * y;

    PrintArray(A, n);

    return z * y;
}

int main(void)
{
    int A[] = {1, 2, 3, 4, 5};
    int n = sizeof(A) / sizeof(A[0]);

    X(A, n, 1, 0);

    return 0;
}
```

From main -> X(A, 5, 1, 0)

1st call
n = 5, x = 1, i = 0
i != n (0 != 5)
z = A[i] = A[0] = 1
y = X(A, 5, 1, 1)
A[i] = A[0] = x * y
PrintArray
return z*y

2nd call
n = 5, x = 1, i = 1
i != n (1 != 5)
z = A[i] = A[1] = 2
y = X(A, 5, 2, 2)
A[i] = A[1] = x * y
PrintArray
return z*y

3rd call
n = 5, x = 2, i = 2
i != n (2 != 5)
z = A[i] = A[2] = 3
y = X(A, 5, 6, 3)
A[i] = A[2] = x * y
PrintArray
return z*y

4th call
n = 5, x = 6, i = 3
i != n (3 != 5)
z = A[i] = A[3] = 4
y = X(A, 5, 24, 4)
A[i] = A[3] = x * y
PrintArray
return z*y

5th call
n = 5, x = 24, i = 4
i != n (4 != 5)
z = A[i] = A[4] = 5
y = X(A, 5, 120, 5)
A[i] = A[4] = x * y
PrintArray
return z*y

6th call
n = 5, x = 120, i = 5
i != n (5 == 5)
return 1

3rd call
n = 5, x = 2, i = 2
i != n (2 != 5)
z = A[i] = A[2] = 3
y = X(A, 5, 6, 3)   y = 20
A[i] = A[2] = x * y   A[2] = x * y = 2 * 20
PrintArray   {1, 2, 40, 30, 24}
return z*y   return z*y (3*20) => 60

4th call
n = 5, x = 6, i = 3
i != n (3 != 5)
z = A[i] = A[3] = 4
y = X(A, 5, 24, 4)   y = 5
A[i] = A[3] = x * y   A[3] = x * y = 6 * 5
PrintArray   {1, 2, 3, 30, 24}
return z*y   return z*y (4*5) => 20

5th call
n = 5, x = 24, i = 4
i != n (4 != 5)
z = A[i] = A[4] = 5
y = X(A, 5, 120, 5)   y = 1
A[i] = A[4] = x * y   A[4] = x * y = 24 * 1
PrintArray   {1, 2, 3, 4, 24}
return z*y   return z*y (5*1) => 5

6th call
n = 5, x = 120, i = 5
i != n (5 == 5)
return 1

1st call
n = 5, x = 1, i = 0
i != n (0 != 5)
z = A[i] = A[0] = 1
y = X(A, 5, 1, 1)   y = 120
A[i] = A[0] = x * y   A[0] = x * y = 1 * 120
PrintArray   {120, 60, 40, 30, 24}
return z*y   return z*y (1*120) => 120

2nd call
n = 5, x = 1, i = 1
i != n (1 != 5)
z = A[i] = A[1] = 2
y = X(A, 5, 2, 2)   y = 60
A[i] = A[1] = x * y   A[1] = x * y = 1 * 60
PrintArray   {1, 60, 40, 30, 24}
return z*y   return z*y (2*60) => 120

main() does use the return value of the recursive function so the program ends.

The result of all printing is

```
1  2  3  4  24
1  2  3  30  24
1  2  40  30  24
1  60  40  30  24
120  60  40  30  24
```

Asymptotic Notation

Suppose you are on a game show and are given the following challenge with the prize being a million dollars.

You are shown 2 identical boxes and are told

Box A contains between 10 and 20 bugs

Box B contains between 30 and 40 bugs

To win, you must pick a box and eat all of the bugs inside and we are going to assume that eating less bugs is better than eating more bugs.

Asymptotic Notation

Let's examine what we know about Box A

      Box A contains between 10 and 20 bugs

We can say that Box A cannot have less than 5 bugs.

That would make 5 a lower bound on the number of bugs in Box A.

This lower bound does not really help us make a decision.

We would rather take 10 as the lower bound.

While 5 and 10 are both valid lower bounds, we would rather use 10 since it is closer than 5 to the actual number of bugs in the box.

Asymptotic Notation

Let's examine what we know about Box A

     Box A contains between 10 and 20 bugs

We can say that Box A cannot have more than 50 bugs.

That would make 50 an upper bound on the number of bugs in Box A.

This upper bound does not really help us make a decision.

We would rather take 20 as the upper bound.

While 50 and 20 are both valid upper bounds, we would rather use 20 since it is closer than 50 to the actual number of bugs in the box.

Asymptotic Notation

The actual number of bugs in Box A must be between our lower bound of 10 and our upper bound of 20.

But…

we also have Box B to analyze.

Remember that Box B contains between 30 and 40 bugs.

Using the same technique we used to analyze Box B, we can determine that Box B has an upper bound of 40 and a lower bound of 30.

Asymptotic Notation

Now we need to analyze the upper and lower bounds of the number of bugs you will need to eat in the best and worst case scenarios.

Best case scenario is you pick Box A - the actual number of bugs in Box A must be between our lower bound of 10 and our upper bound of 20.

Worst case scenario is you pick Box B – the actual number of bugs in Box B must be between our lower bound of 30 and our upper bound of 40.

The best case scenario has both lower and upper bounds.

The worst case scenario has both lower and upper bounds.

Asymptotic Notation

Since we cannot distinguish Box A from Box B, then we are faced with

      overall worst case scenario is the upper bound of all cases

        50 bugs

      overall best case scenario is the lower bound of all cases

        10 bugs

Can we get any closer to the actual number of bugs you have to eat to get that million dollars?

Asymptotic Notation

Prove this is true

$\log_2 n$ is $O(\log_8 n)$

This is saying that $\log_2 n$ has an asymptotic upper bound of $O(\log_8 n)$

This means that for large enough value of n, the running time is at most $k \cdot \log_8 n$ for some constant k. $\log_8 n$ is an upper bound.

Can we come up with a k to multiply $\log_8 n$ by so that it always bounds $\log_2 n$?

$\log_2(x)$

$\log_8(x)$

Run time

100

80

60

40

20

0   20   40   60   80   100   120   140

Array Elements
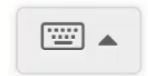
powered by
desmos

Asymptotic Notation

Prove this is true

$\log_2 n$ is $O(\log_8 n)$

Let's remember that

$$\log_a n = \frac{\log_b n}{\log_b a}$$

So $\log_8 n$ can be written as

$$\log_8 n = \frac{\log_2 n}{\log_2 8} \text{ which is}$$

$$\log_8 n = \frac{1}{\log_2 8} \log_2 n$$

So $\log_8 n$ is just $\log_2 n$ times a constant

Asymptotic Notation

Prove this is true $\log_2 n$ is $O(\log_8 n)$

$$\log_8 n = \frac{1}{\log_2 8} \log_2 n$$

So $\log_8 n$ is just $\log_2 n$ times a constant

When we find two functions that differ by a constant multiplier, we know that we can always find a k to serve as the upper bound.

Generally, we can effectively ignore constants in asymptotic notation and treat the functions as equivalent.

Asymptotic Notation
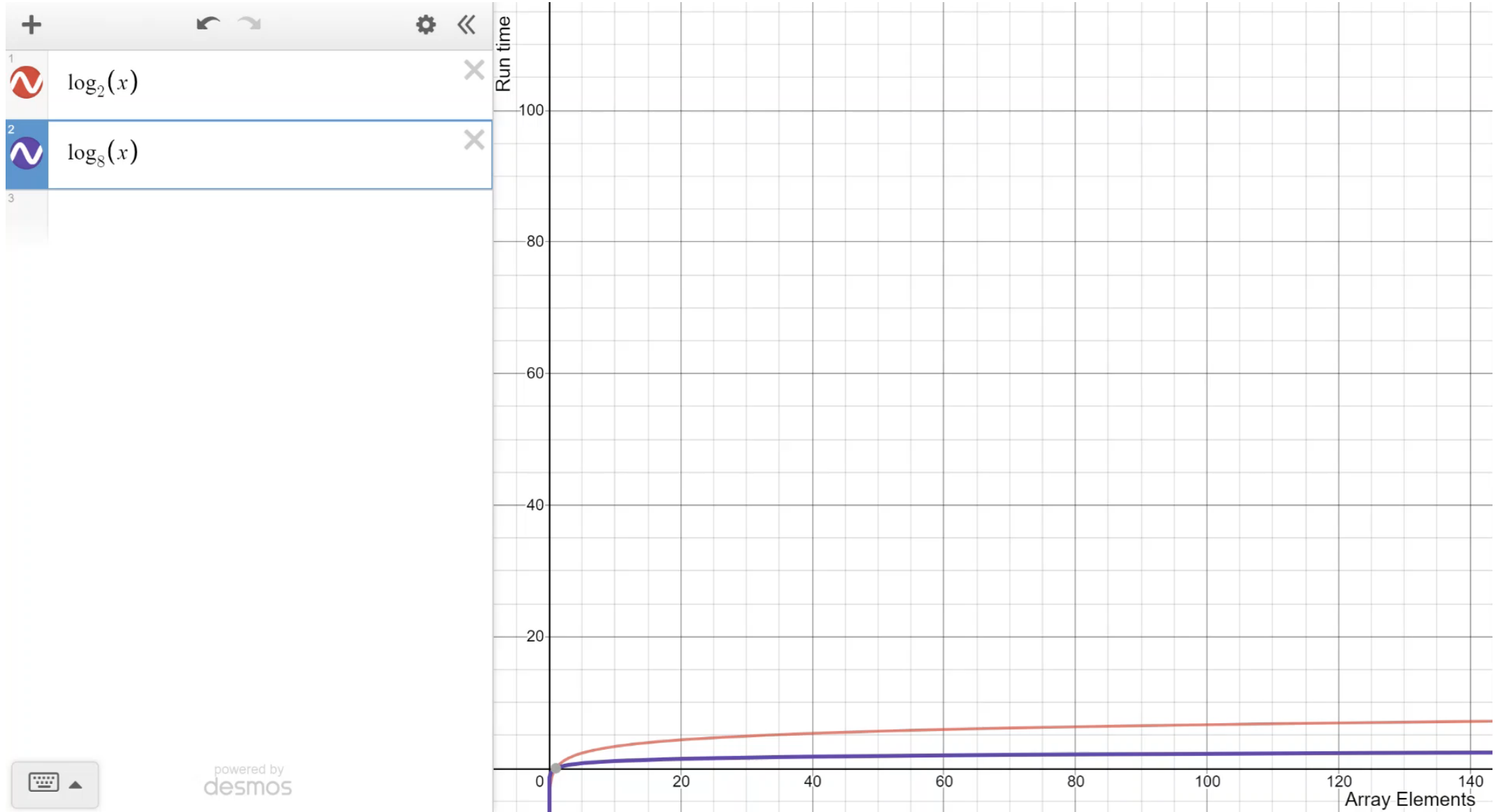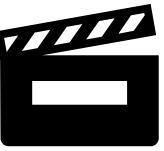
Prove this is true $\log_2 n$ is $\Omega(\log_8 n)$

This is saying that $\log_2 n$ has an asymptotic lower bound of $\Omega(\log_8 n)$

This means that for large enough value of n, the running time is at least $k \cdot \log_8 n$ for some constant k. $\log_8 n$ is an lower bound.

Can we come up with a k to multiply $\log_8 n$ by so that it always lower bounds $\log_2 n$?

Asymptotic Notation

Prove this is true: $\log_2 n$ is $\Omega(\log_8 n)$

We have already rewritten $\log_8 n$ as

$\log_8 n = \dfrac{1}{\log_2 8} \log_2 n$

That constant, $\dfrac{1}{\log_2 8}$, is already a value for k that shows $\log_8 n$ to be a lower bound.
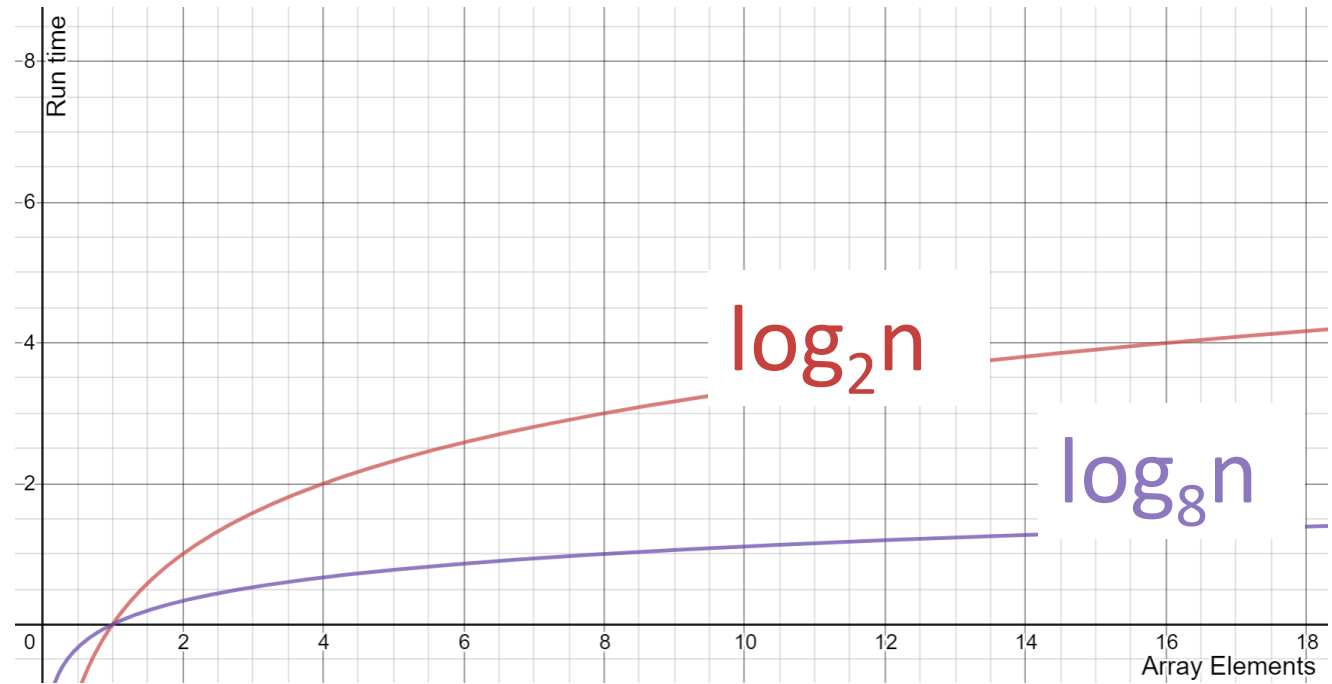
Asymptotic Notation

Prove this is true: $\log_2 n$ is $\Theta(\log_8 n)$

If $\log_2 n$ is $\Theta(\log_8 n)$, then $\log_2 n$ is "tightly bound" by $\Theta(\log_8 n)$

for a large enough value of n, the running time is at least $k_1 \cdot \log_8 n$ and at most $k_2 \cdot \log_8 n$ for some constants k_1 and k_2

We have already found k_1 and k_2 that satisfies those constraints when we proved O and Ω notation.

Since we established $\log_2 n$ is $O(\log_8 n)$ and $\Omega(\log_8 n)$, we can conclude that it is also $\Theta(\log_8 n)$.

Divide and Conquer
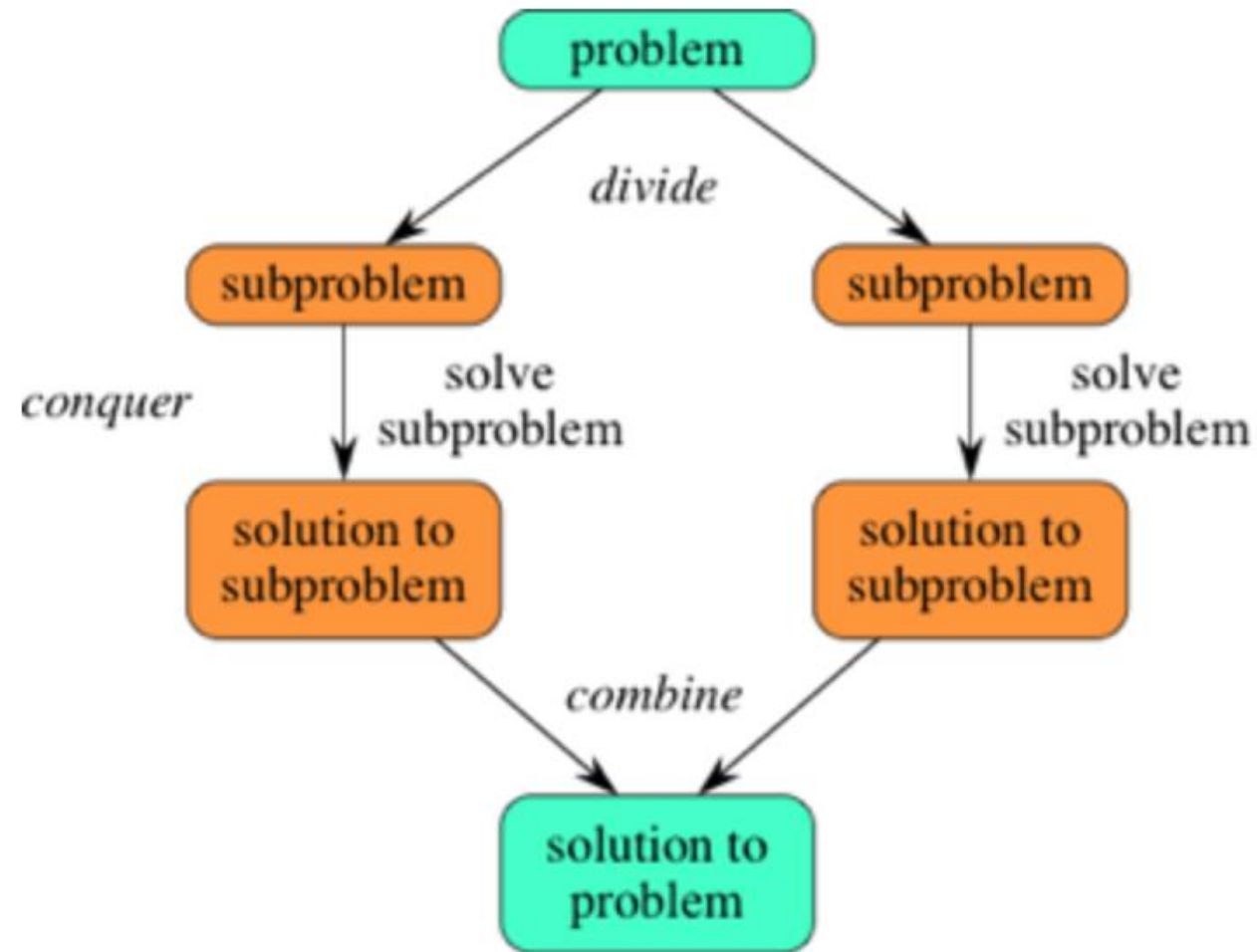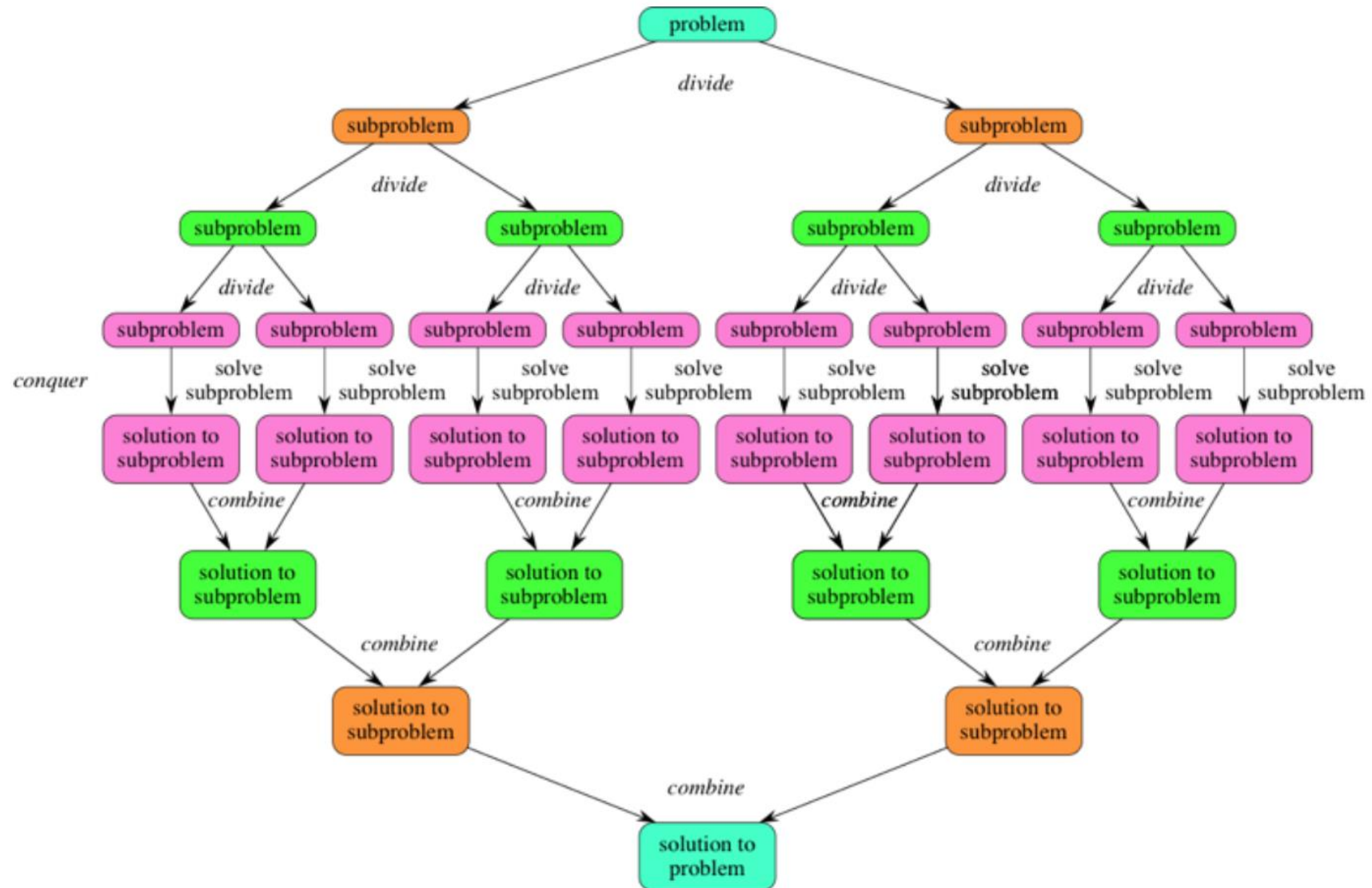
Divide and Conquer

Divide and Conquer
Merge Sort

When using divide-and-conquer to sort, we need to decide what our subproblems are going to look like.

The full problem is to sort an entire array.

The subproblem is to sort a subarray.

Divide and Conquer
Merge Sort

To sort `A[p .. r]`

Divide by splitting into two subarrays `A[p .. q]` and `A[q+1 .. r]` where q is the halfway point of `A[p .. r]`

Conquer by recursively sorting the two subarrays `A[p .. q]` and `A[q+1 .. r]`

Combine by merging the two sorted subarrays `A[p .. q]` and `A[q+1 .. r]` to produce a single sorted subarray `A[p .. r]`.

The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted – can't divide 1 value.

divide

divide

divide

merge

merge

merge

0	1	2	3	4	5	6	7
14	7	3	12	9	11	6	2

0	1	2	3
14	7	3	12

4	5	6	7
9	11	6	2

0	1
14	7

2	3
3	12

4	5
9	11

6	7
6	2

0	1	2	3	4	5	6	7
14	7	3	12	9	11	6	2

0	1
7	14

2	3
3	12

4	5
9	11

6	7
2	6

0	1	2	3
3	7	12	14

4	5	6	7
2	6	9	11

0	1	2	3	4	5	6	7
2	3	6	7	9	11	12	14
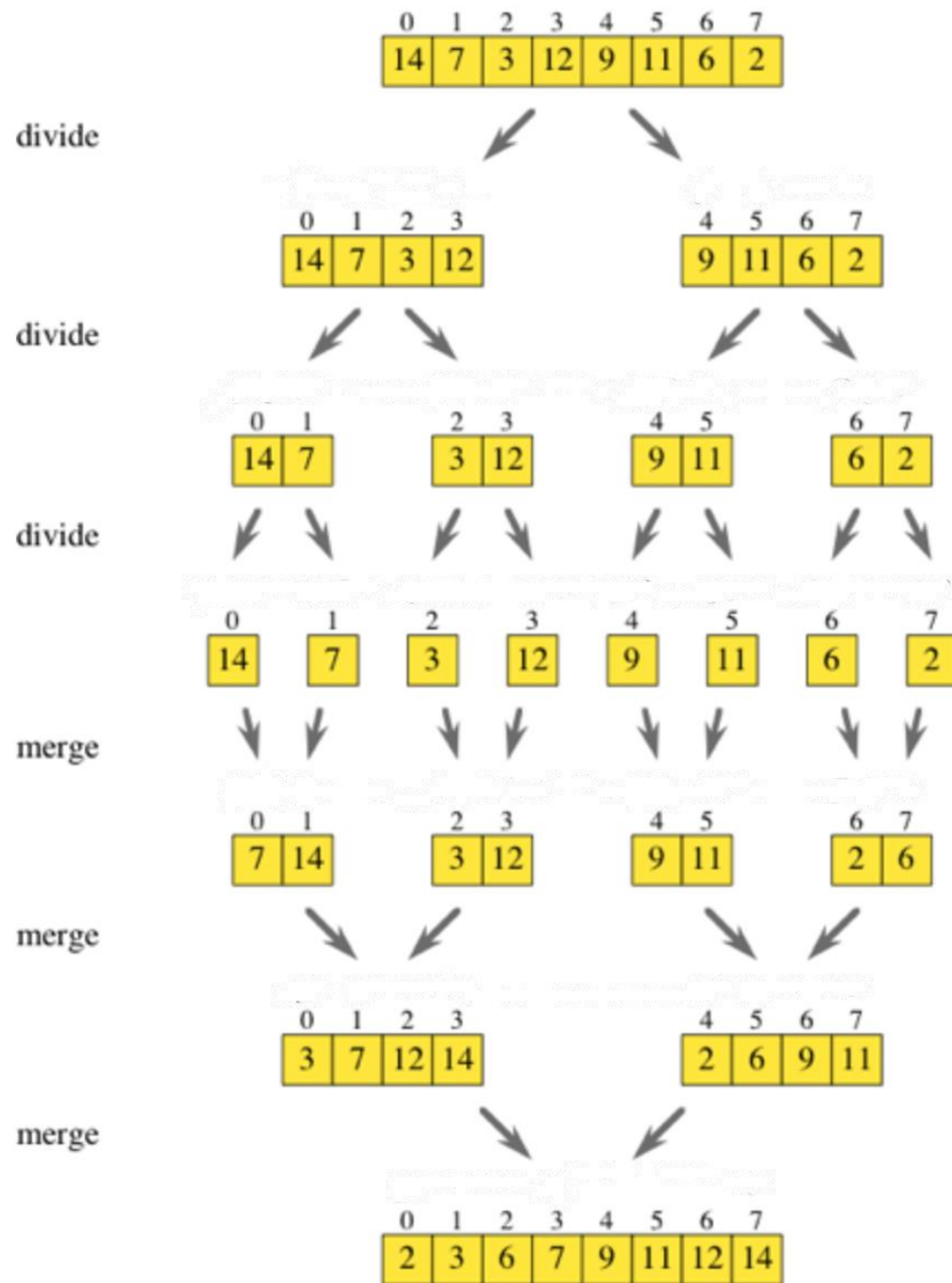
Analysis of Merge Sort

Let's examine the merge function from Merge Sort.

The merge function merges two sorted arrays into a single sorted array.

If the two subarrays have a total of n elements, then how many times did we examine each element in order to accomplish the merge?

We have to examine each of the elements in order to merge them together which gives us a merging time of $\Theta(n)$.

Analysis of Merge Sort

Given that the merge function runs in $\Theta(n)$ time when merging n elements, how do demonstrate that Merge Sort runs in $\Theta(n\log_2 n)$ time?

We start by thinking about the three parts of divide-and-conquer and how to account for their running times.

We assume that we are sorting a total of n elements in the entire array.

Analysis of Merge Sort

The divide step takes constant time, regardless of the subarray size.

> The divide step just computes the midpoint q of the indices p and r. (middle = (left+right)/2)

> In big-Θ notation, we indicate constant time by $\Theta(1)$.

Using 1 with Θ indicates no growth

The conquer step, where we recursively sort two subarrays of approximately $\frac{n}{2}$ elements each, takes some amount of time, but we'll account for that time when we consider the subproblems.

The combine step merges a total of n elements, taking $\Theta(n)$ time.

Analysis of Merge Sort

Let's put the divide and combine steps together.

The $\Theta(1)$ running time for the divide step is a low-order term when compared with the $\Theta(n)$ running time of the combine step. As n increases, the amount of constant time needed for the divide step will be trivial.

So let's eliminate the divide time as a separate consideration and just take the divide and combine steps together as $\Theta(n)$ time.

To make things more concrete, let's say that the divide and combine steps together take cn time for some constant c.

Analysis of Merge Sort

To keep things reasonably simple, let's assume that if $n > 1$, then n is always even.

Why?

So that when we need to think about $\frac{n}{2}$, it's an integer – integer division does not lose anything.

Accounting for the case in which n is odd doesn't change the result in terms of big-Θ notation.

Analysis of Merge Sort

So now we can think of the running time of Merge Sort on an *n*-element subarray as being

the sum of

      twice the running time of Merge Sort on an $\frac{n}{2}$-element subarray

        (for the conquer step)

plus

      cn

        (for the divide and combine steps—really for just the merging).

Analysis of Merge Sort

Now we have to figure out the running time of two recursive calls on $\frac{n}{2}$

Each of these two recursive calls takes twice of the running time of Merge Sort on an $\frac{n}{4}$-element subarray (because we have to halve $\frac{n}{2}$) plus $\frac{cn}{2}$ to merge.

We have two subproblems of size $\frac{n}{2}$ and each takes $\frac{cn}{2}$ time to merge.

The total time we spend merging for subproblems of size $\frac{n}{2}$ is $2 * \frac{cn}{2} = cn$

Analysis of Merge Sort
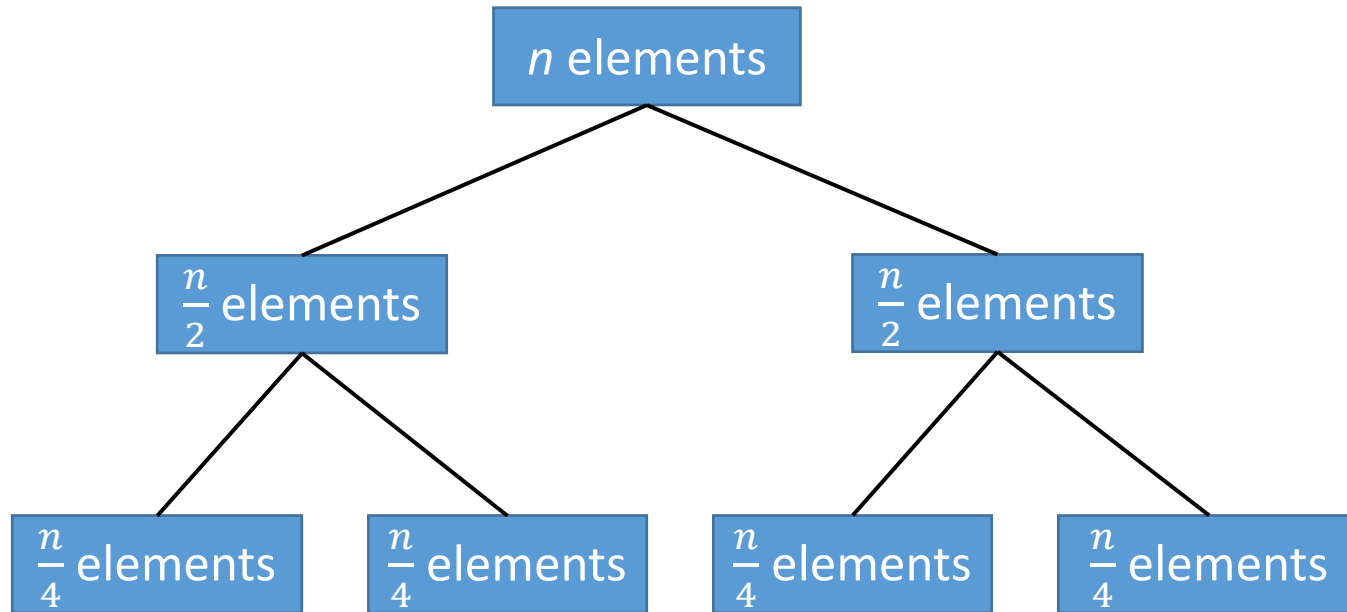
Total merge time for array of this size

cn

$2 * \dfrac{cn}{2} = cn$

$4 * \dfrac{cn}{4} = cn$

What would be the merge time for the next level of $\dfrac{n}{8}$? $8 * \dfrac{cn}{8} = cn$

Analysis of Merge Sort

As the subproblems/arrays get smaller, the number of subproblems doubles at each "level" of the recursion, but the merging time halves.

The doubling and halving cancel each other out so the total merging time is *cn* at each level of recursion.

Subproblem size

Total merging time for all subproblems of this size

n cn

$n/2$ $n/2$ $2 \cdot cn/2 = cn$

$n/4$ $n/4$ $n/4$ $n/4$ $4 \cdot cn/4 = cn$

$n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $8 \cdot cn/8 = cn$
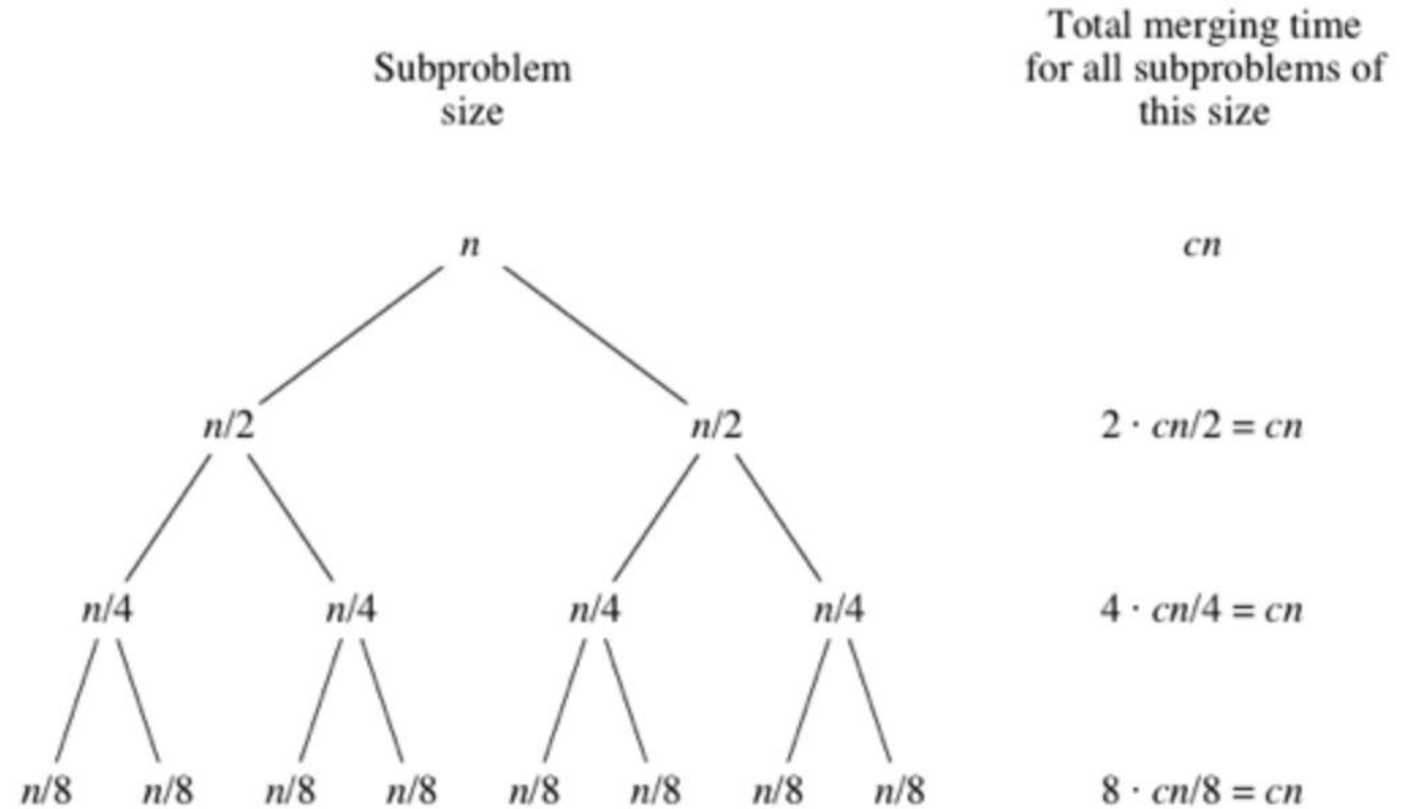
Analysis of Merge Sort

Eventually, we get down to subproblems of size 1 (the base case)

We have to spend $\Theta(1)$ time to sort subarrays of size 1
     we have to test whether $p < r$ (left < right)
     this test takes time but that time is constant/the same every time
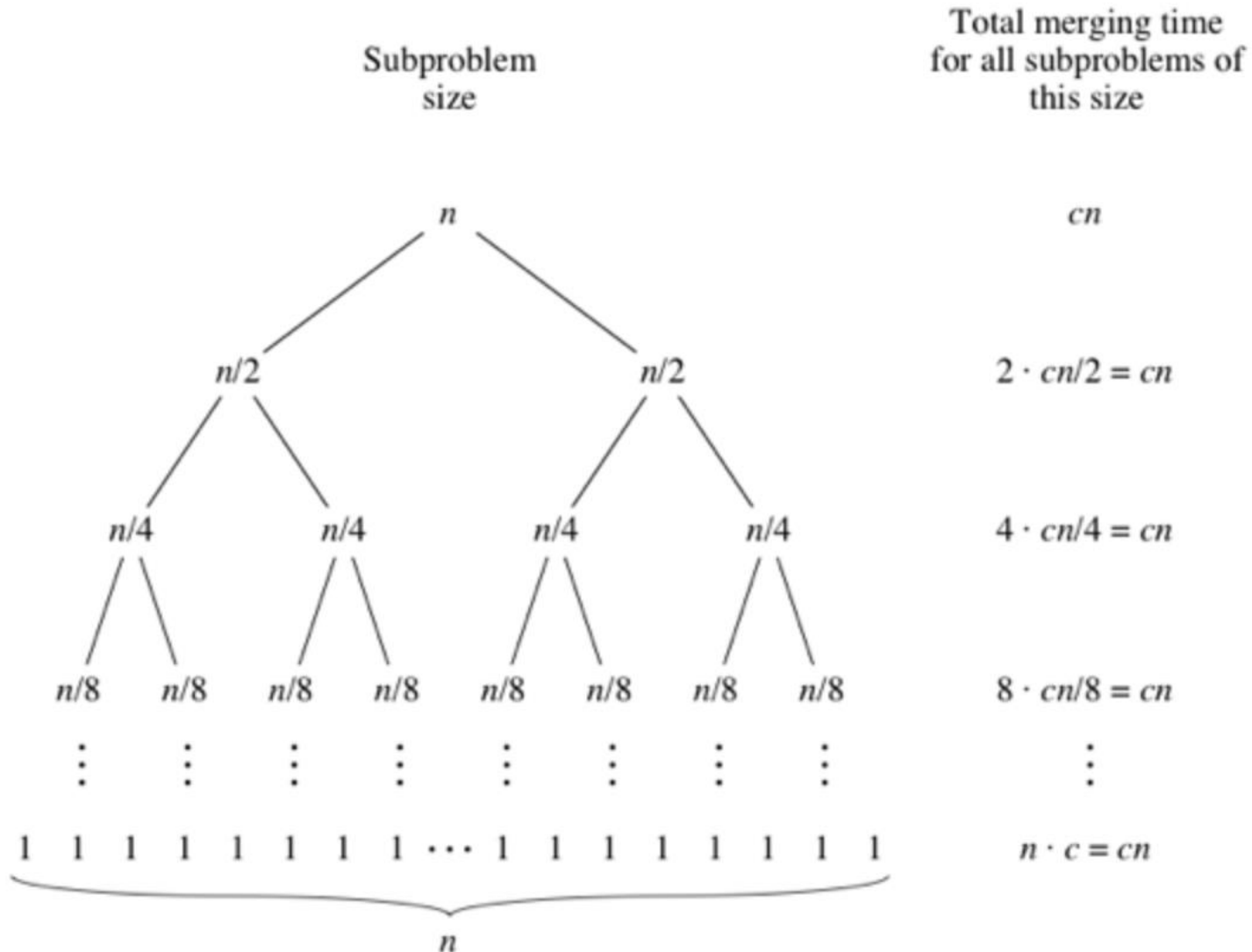
How many subarrays of size 1 are there?
     Since we started with n elements, there must be n base cases.
     Since each base case takes $\Theta(1)$ time, the base cases take cn time

Analysis of Merge Sort

Subproblem size

Total merging time for all subproblems of this size

n — cn

$n/2$ — $n/2$ — $2 \cdot cn/2 = cn$

$n/4$ — $n/4$ — $n/4$ — $n/4$ — $4 \cdot cn/4 = cn$

$n/8$ — $n/8$ — $n/8$ — $n/8$ — $n/8$ — $n/8$ — $n/8$ — $n/8$ — $8 \cdot cn/8 = cn$

1 1 1 1 1 1 1 1 $\cdots$ 1 1 1 1 1 1 1 1 — $n \cdot c = cn$

n

The total time merging in Merge Sort is the sum of the merging times for all the levels.

If there are Z levels in the tree, then the total merging time is

Z * cn

So what is Z?

Does this pattern look familiar?

Analysis of Merge Sort

$cn\log_2 n$

Subproblem size	Total merging time for all subproblems of this size
n	cn
$n/2$ $n/2$	$2 \cdot cn/2 = cn$
$n/4$ $n/4$ $n/4$ $n/4$	$4 \cdot cn/4 = cn$
$n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$	$8 \cdot cn/8 = cn$
1 1 1 1 1 1 1 1 $\cdots$ 1 1 1 1 1 1 1 1	$n \cdot c = cn$
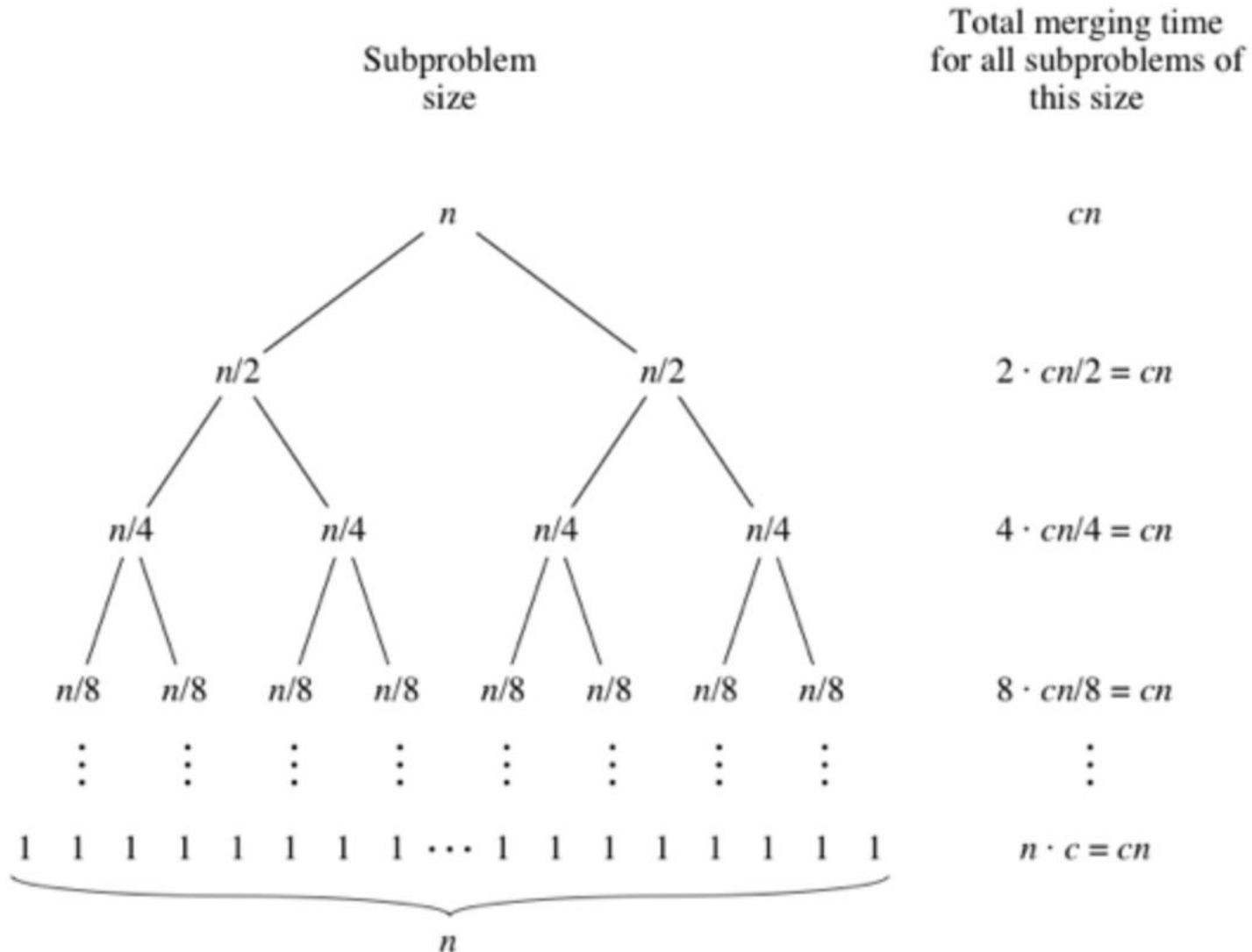
n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6

Analysis of Merge Sort

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6

You could argue that this runtime should be

$cn\log_2 n + 1$

Why + 1?
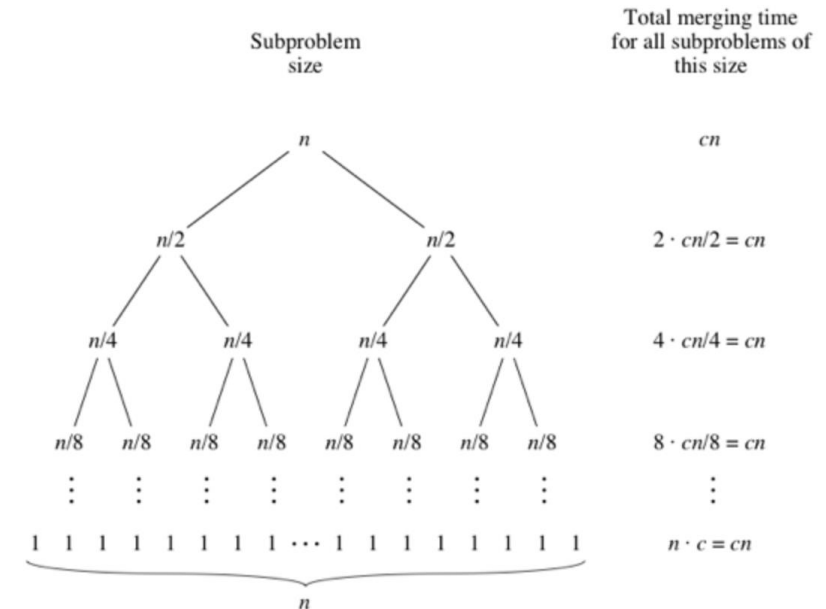
What happens when we have an array of 1 element?

$cn\log_2 n$ when $n = 1$ is 0

Does it take 0 time to sort an array of 1 element?

You could argue that it takes some time to check for that 1 element.

Subproblem size	Total merging time for all subproblems of this size
n	cn
$n/2$ $n/2$	$2 \cdot cn/2 = cn$
$n/4$ $n/4$ $n/4$ $n/4$	$4 \cdot cn/4 = cn$
$n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$	$8 \cdot cn/8 = cn$
1 1 1 1 1 1 1 1 $\cdots$ 1 1 1 1 1 1 1 1	$n \cdot c = cn$

Analysis of Merge Sort

So we have calculated that the runtime of Merge Sort to be

$$cn\log_2 n \qquad \text{or} \qquad cn\log_2 n + 1$$

Using Θ notation allows us to discard the constant coefficient c and the lower order term (+ 1) if we included it . This gives us a runtime of

$$\Theta(n\log_2 n)$$

OLQ3

Online Quiz 3 (OLQ3) will cover the recursive divide and conquer logic of MergeSort.

You will be given the code for MergeSort and you will be given an array.

You will need to trace the recursive calls so that you can list each call to the **merge function** in the correct orders and its parameters (left, middle, right).

You will also need to list the contents of the left array (L) and the right array (R) after they are created in Merge – Part 1.

You will also need to list the contents of the main array **after** each call to the merge function completes.

OLQ3

For example, in lecture, the array was `{12, 11, 5, 13, 7, 6}`

You will need to provide the following information.

Step	left	middle	right	L	R	array
A	0	0	1	12	11	11,12,5,13,7,6
B	0	1	2	11,12	5	5,11,12,13,7,6
C	3	3	4	13	7	5,11,12,7,13,6
D	3	4	5	7,13	6	5,11,12,6,7,13
E	0	2	5	5,11,12	6,7,13	5,6,7,11,12,13