Data Structures and Algorithms

CS245-2017S-11

Sorting in $\Theta(n \lg n)$

David Galles

Department of Computer Science
University of San Francisco

11-0: Merge Sort – Recursive Sorting

- Base Case:
  - A list of length 1 or length 0 is already sorted
- Recursive Case:
  - Split the list in half
  - Recursively sort two halves
  - Merge sorted halves together

Example: 5 1 8 2 6 4 3 7

11-1: Merging

- Merge lists into a new temporary list, T
- Maintain three pointers (indices) i, j, and n
  - i is index of left hand list
  - j is index of right hand list
  - n is index of temporary list T
- If $A[i] < A[j]$
  - $T[n] = A[i]$, increment n and i
- else
  - $T[n] = A[j]$, increment n and j

Example: 1 2 5 8 and 3 4 6 7

11-2: $\Theta()$ for Merge Sort

$T(0) = c_1$ for some constant c_1

$T(1) = c_2$ for some constant c_2

$T(n) = nc_3 + 2T(n/2)$ for some constant c_3

$$T(n) = nc_3 + 2T(n/2)$$

11-3: $\Theta()$ for Merge Sort

$T(0) = c_1$ for some constant c_1

$T(1) = c_2$ for some constant c_2

$T(n) = nc_3 + 2T(n/2)$ for some constant c_3

$$
\begin{aligned}
T(n) &= nc_3 + 2T(n/2) \\
&= nc_3 + 2(n/2c_3 + 2T(n/4)) \\
&= 2nc_3 + 4T(n/4)
\end{aligned}
$$

11-4: $\Theta()$ for Merge Sort

$T(0) = c_1$ for some constant c_1

$T(1) = c_2$ for some constant c_2

$T(n) = nc_3 + 2T(n/2)$ for some constant c_3

$$\begin{aligned}
T(n) &= nc_3 + 2T(n/2) \\
&= nc_3 + 2(n/2c_3 + 2T(n/4)) \\
&= 2nc_3 + 4T(n/4) \\
&= 2nc_3 + 4(n/4c_3 + 2T(n/8)) \\
&= 3nc_3 + 8T(n/8))
\end{aligned}$$

11-5: $\Theta()$ for Merge Sort

$T(0) = c_1$ for some constant c_1

$T(1) = c_2$ for some constant c_2

$T(n) = nc_3 + 2T(n/2)$ for some constant c_3

$$
\begin{aligned}
T(n) &= nc_3 + 2T(n/2) \\
&= nc_3 + 2(n/2c_3 + 2T(n/4)) \\
&= 2nc_3 + 4T(n/4) \\
&= 2nc_3 + 4(n/4c_3 + 2T(n/8)) \\
&= 3nc_3 + 8T(n/8)) \\
&= 3nc_3 + 8(n/8c_3 + 2T(n/16)) \\
&= 4nc_3 + 16T(n/16)
\end{aligned}
$$

11-6: $\Theta()$ for Merge Sort

$T(0) = c_1$ for some constant c_1

$T(1) = c_2$ for some constant c_2

$T(n) = nc_3 + 2T(n/2)$ for some constant c_3

$$
\begin{aligned}
T(n) &= nc_3 + 2T(n/2) \\
&= nc_3 + 2(n/2c_3 + 2T(n/4)) \\
&= 2nc_3 + 4T(n/4) \\
&= 2nc_3 + 4(n/4c_3 + 2T(n/8)) \\
&= 3nc_3 + 8T(n/8)) \\
&= 3nc_3 + 8(n/8c_3 + 2T(n/16)) \\
&= 4nc_3 + 16T(n/16) \\
&= 5nc_3 + 32T(n/32)
\end{aligned}
$$

11-7: $\Theta()$ for Merge Sort

$T(0) = c_1$ for some constant c_1

$T(1) = c_2$ for some constant c_2

$T(n) = nc_3 + 2T(n/2)$ for some constant c_3

$$\begin{aligned}
T(n) &= nc_3 + 2T(n/2) \\
&= nc_3 + 2(n/2c_3 + 2T(n/4)) \\
&= 2nc_3 + 4T(n/4) \\
&= 2nc_3 + 4(n/4c_3 + 2T(n/8)) \\
&= 3nc_3 + 8T(n/8)) \\
&= 3nc_3 + 8(n/8c_3 + 2T(n/16)) \\
&= 4nc_3 + 16T(n/16) \\
&= 5nc_3 + 32T(n/32) \\
&= knc_3 + 2^kT(n/2^k)
\end{aligned}$$

11-8: $\Theta()$ for Merge Sort

$$T(0) = c_1$$
$$T(1) = c_2$$
$$T(n) = knc_3 + 2^k T(n/2^k)$$

Pick a value for k such that $n/2^k = 1$:

$$\begin{aligned}
n/2^k &= 1 \\
n &= 2^k \\
\lg n &= k \\
T(n) &= (\lg n)nc_3 + 2^{\lg n} T(n/2^{\lg n}) \\
&= c_3 n \lg n + nT(n/n) \\
&= c_3 n \lg n + nT(1) \\
&= c_3 n \lg n + c_2 n \\
&\in O(n \lg n)
\end{aligned}$$

11-9: $\Theta()$ for Merge Sort

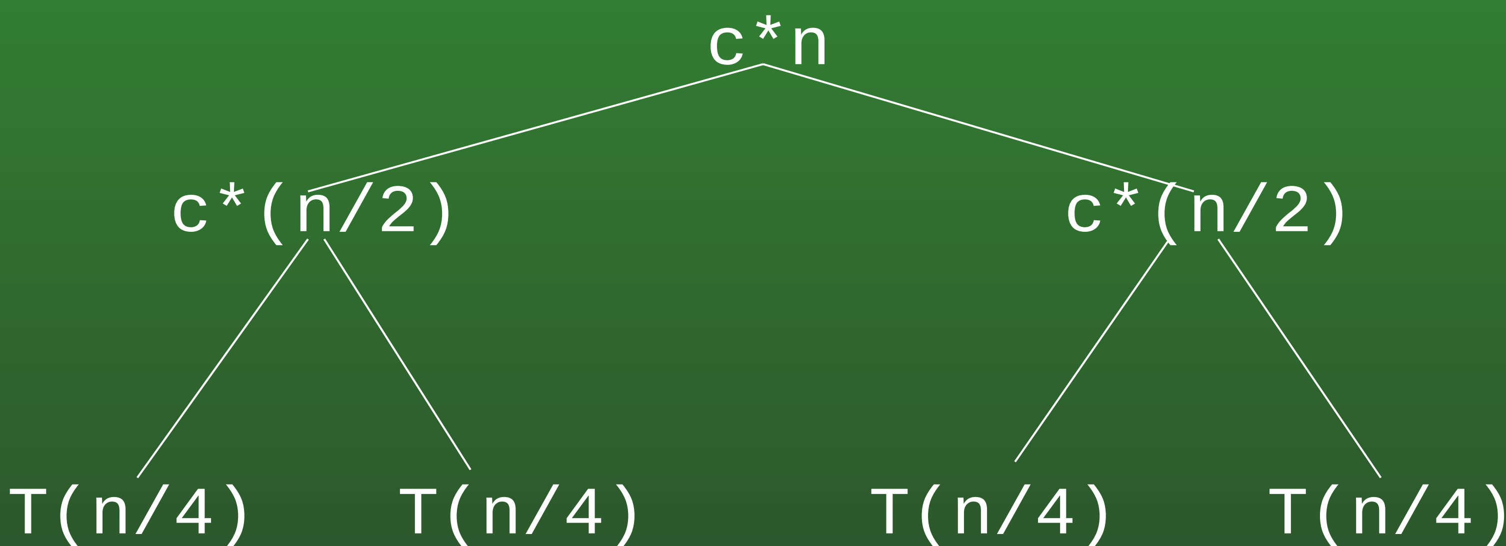
T(n)

11-10: $\Theta()$ for Merge Sort

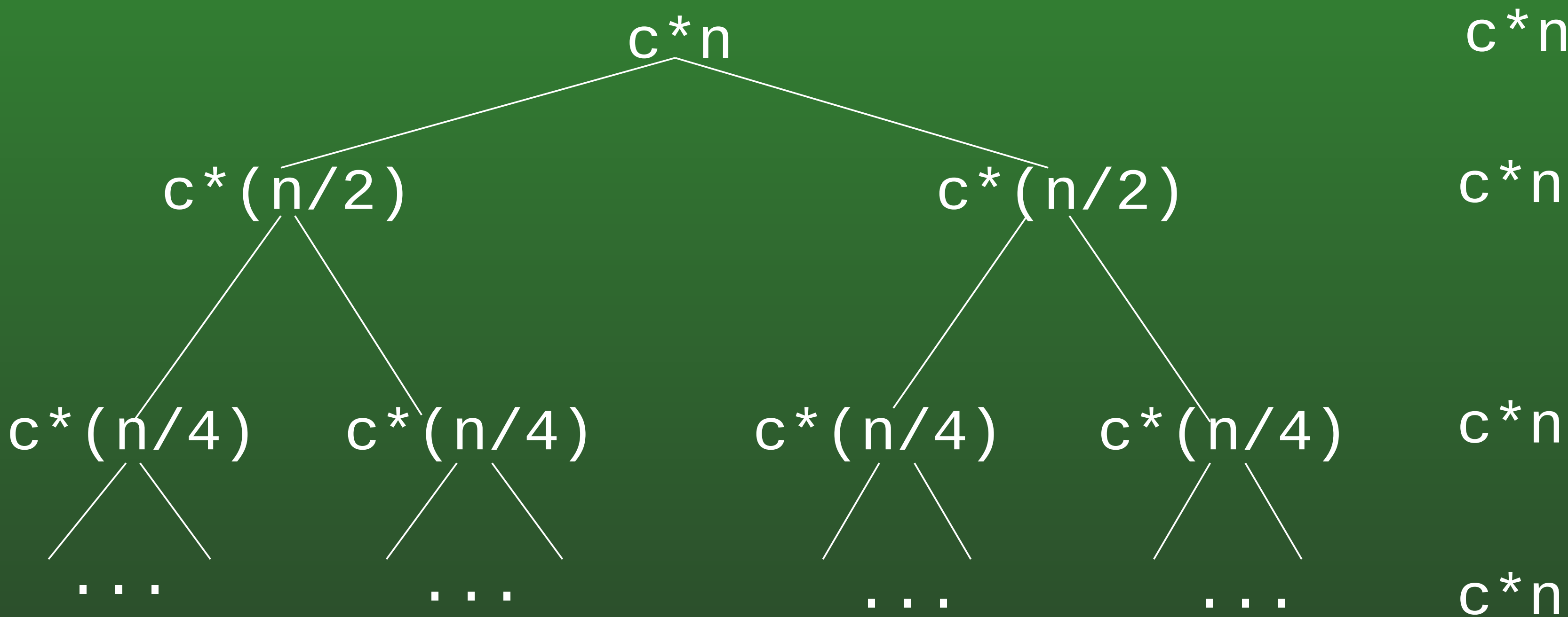
c*n

T(n/2) T(n/2)

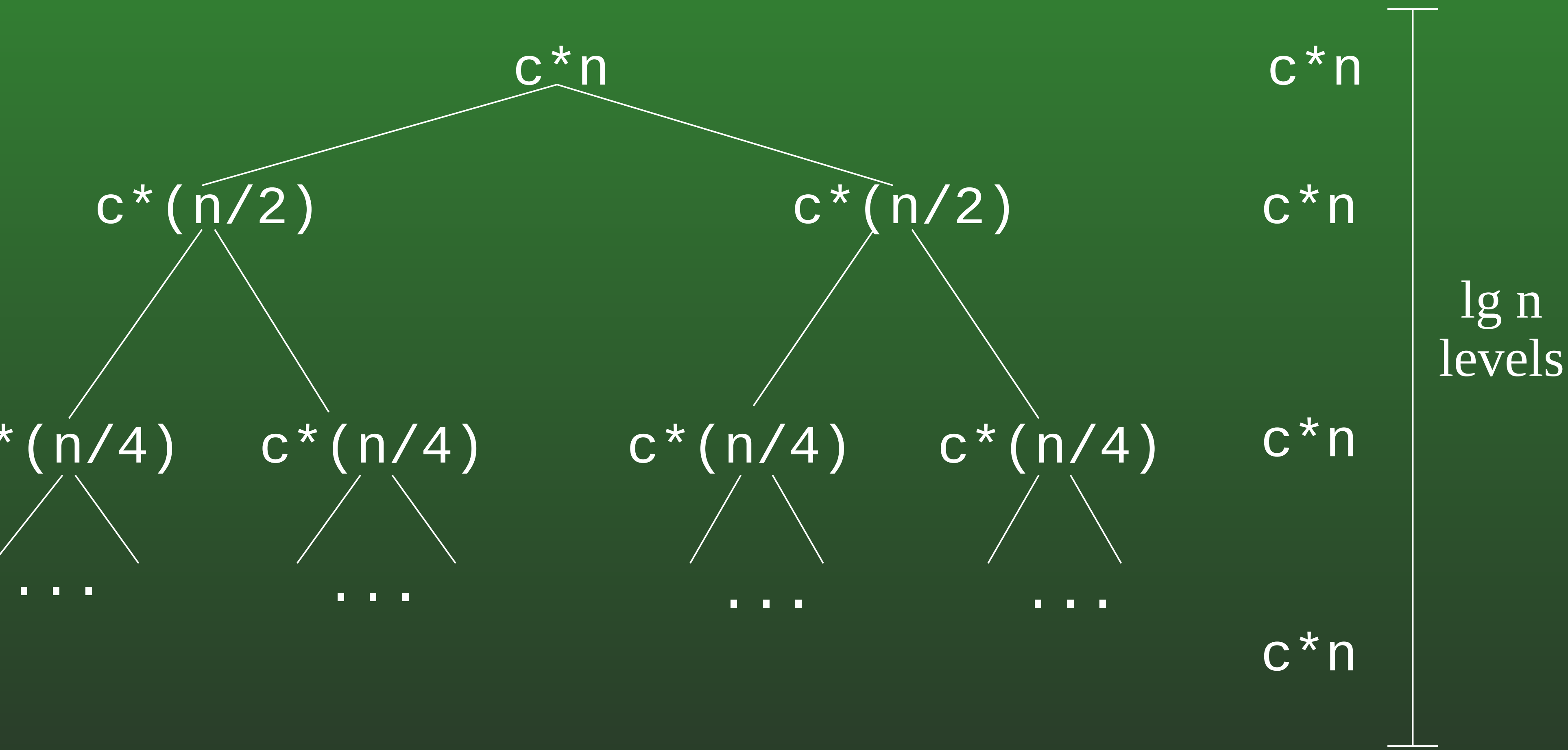
11-11: $\Theta()$ for Merge Sort



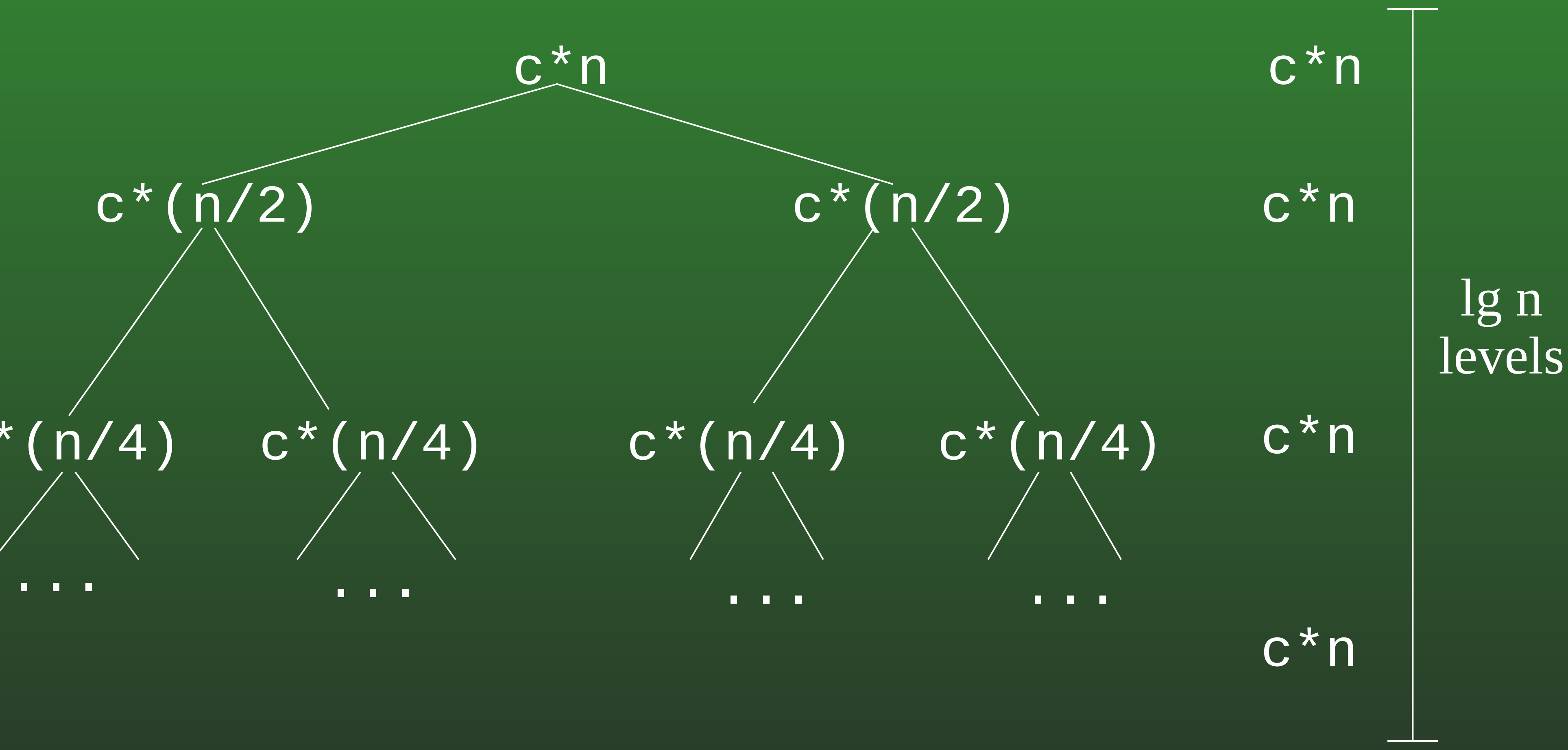
11-12: $\Theta()$ for Merge Sort



11-13: $\Theta()$ for Merge Sort



11-14: $\Theta()$ for Merge Sort



Total time = c*n lg n

Θ(n lg n)

11-15: $\Theta()$ for Merge Sort

$T(0) = c_1$ for some constant c_1

$T(1) = c_2$ for some constant c_2

$T(n) = nc_3 + 2T(n/2)$ for some constant c_3

$T(n) = aT(n/b) + f(n)$

$a = 2, b = 2, f(n) = n$

$n^{\log_b a} = n^{\log_2 2} = n \in \Theta(n)$

By second case of the Master Method,

$T(n) \in \Theta(n \lg n)$

11-16: Divide & Conquer

Merge Sort:

- Divide the list two parts
  - No work required – just calculate midpoint
- Recursively sort two parts
- Combine sorted lists into one list
  - Some work required – need to merge lists

11-17: Divide & Conquer

Quick Sort:

- Divide the list two parts
  - Some work required – Small elements in left sublist, large elements in right sublist
- Recursively sort two parts
- Combine sorted lists into one list
  - No work required!

11-18: Quick Sort

- Pick a pivot element
- Reorder the list:
  - All elements $<$ pivot
  - Pivot element
  - All elements $>$ pivot
- Recursively sort elements $<$ pivot
- Recursively sort elements $>$ pivot

Example: 3 7 2 8 1 4 6

11-19: Quick Sort - Partitioning

Basic Idea:

- Swap pivot elememt out of the way (we'll swap it back later)
- Maintain two pointers, i and j
  - i points to the beginning of the list
  - j points to the end of the list
- Move i and j in to the middle of the list – ensuring that all elements to the left of i are $<$ the pivot, and all elememnts to the right of j are greater than the pivot
- Swap pivot element back to middle of list

11-20: Quick Sort - Partitioning

Pseudocode:

- Pick a pivot index
- Swap A[pivotindex] and A[high]
- Set $i \leftarrow low$, $j \leftarrow$ high-1
- while $(i <= j)$
  - while $A[i] < A[\text{pivot}]$, increment i
  - while $A[j] > A[\text{pivot}]$, decrement i
  - swap $A[i]$ and $A[j]$
  - increment i, decrement j
- swap $A[i]$ and $A[\text{pivot}]$

11-21: $\Theta()$ for Quick Sort

- Coming up with a recurrence relation for quicksort is harder than mergesort
- How the problem is divided depends upon the data
  - Break list into:

    size 0, size $n-1$
    size 1, size $n-2$
    $\ldots$
    size $\lfloor (n-1)/2 \rfloor$, size $\lceil (n-1)/2 \rceil$
    $\ldots$
    size $n-2$, size 1
    size $n-1$, size 0

11-22: $\Theta()$ for Quick Sort

Worst case performance occurs when break list into size $n-1$ and size 0

$T(0) = c_1$ for some constant c_1

$T(1) = c_2$ for some constant c_2

$T(n) = nc_3 + T(n-1) + T(0)$ for some constant c_3

$$\begin{aligned} T(n) &= nc_3 + T(n-1) + T(0) \\ &= T(n-1) + nc_3 + c_2 \end{aligned}$$

11-23: $\Theta()$ for Quick Sort

Worst case: $T(n) = T(n-1) + nc_3 + c_2$

$$T(n)$$
$$= T(n-1) + nc_3 + c_2$$

11-24: $\Theta()$ for Quick Sort

Worst case: $T(n) = T(n-1) + nc_3 + c_2$

$$
\begin{aligned}
& T(n) \\
& = T(n-1) + nc_3 + c_2 \\
& = [T(n-2) + (n-1)c_3 + c_2] + nc_3 + c_2 \\
& = T(n-2) + (n + (n-1))c_3 + 2c_2
\end{aligned}
$$

11-25: $\Theta()$ for Quick Sort

Worst case: $T(n) = T(n-1) + nc_3 + c_2$

$$
\begin{aligned}
& T(n) \\
& = T(n-1) + nc_3 + c_2 \\
& = [T(n-2) + (n-1)c_3 + c_2] + nc_3 + c_2 \\
& = T(n-2) + (n + (n-1))c_3 + 2c_2 \\
& = [T(n-3) + (n-2)c_3 + c_2] + (n + (n-1))c_3 + 2c_2 \\
& = T(n-3) + (n + (n-1) + (n-2))c_3 + 3c_2
\end{aligned}
$$

11-26: $\Theta()$ for Quick Sort

Worst case: $T(n) = T(n-1) + nc_3 + c_2$

$$
\begin{aligned}
&T(n) \\
&= T(n-1) + nc_3 + c_2 \\
&= [T(n-2) + (n-1)c_3 + c_2] + nc_3 + c_2 \\
&= T(n-2) + (n + (n-1))c_3 + 2c_2 \\
&= [T(n-3) + (n-2)c_3 + c_2] + (n + (n-1))c_3 + 2c_2 \\
&= T(n-3) + (n + (n-1) + (n-2))c_3 + 3c_2 \\
&= T(n-4) + (n + (n-1) + (n-2) + (n-3))c_3 + 4c_2
\end{aligned}
$$

11-27: $\Theta()$ for Quick Sort

Worst case: $T(n) = T(n-1) + nc_3 + c_2$

$$
\begin{aligned}
&T(n) \\
&= T(n-1) + nc_3 + c_2 \\
&= [T(n-2) + (n-1)c_3 + c_2] + nc_3 + c_2 \\
&= T(n-2) + (n + (n-1))c_3 + 2c_2 \\
&= [T(n-3) + (n-2)c_3 + c_2] + (n + (n-1))c_3 + 2c_2 \\
&= T(n-3) + (n + (n-1) + (n-2))c_3 + 3c_2 \\
&= T(n-4) + (n + (n-1) + (n-2) + (n-3))c_3 + 4c_2 \\
&\ldots \\
&= T(n-k) + (\textstyle\sum_{i=0}^{k-1}(n-i)c_3) + kc_2
\end{aligned}
$$

11-28: $\Theta()$ for Quick Sort

Worst case:

$T(n) = T(n-k) + (\sum_{i=0}^{k-1}(n-i)c_3) + kc_2$

Set $k = n$:

$$
\begin{aligned}
T(n) &= T(n-k) + (\textstyle\sum_{i=0}^{k-1}(n-i)c_3) + kc_2 \\
&= T(n-n) + (\textstyle\sum_{i=0}^{n-1}(n-i)c_3) + kc_2 \\
&= T(0) + (\textstyle\sum_{i=0}^{n-1}(n-i)c_3) + kc_2 \\
&= T(0) + (\textstyle\sum_{i=0}^{n-1} ic_3) + kc_2 \\
&= c_1 + c_3 n(n+1)/2 + kc_2 \\
&\in \Theta(n^2)
\end{aligned}
$$

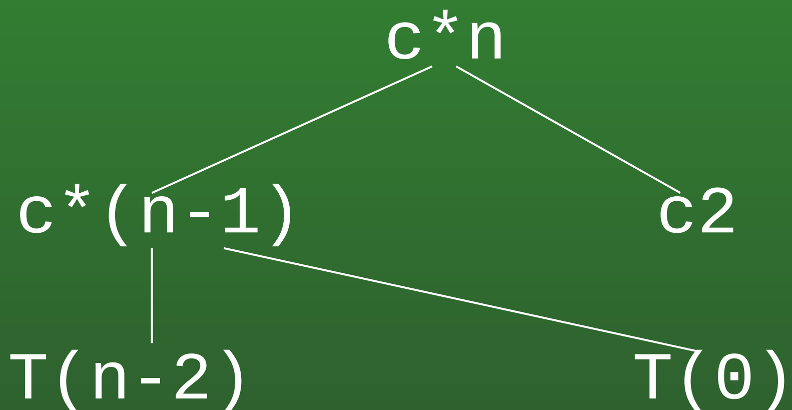
11-29: $\Theta()$ for Quick Sort

```
T(n)
```

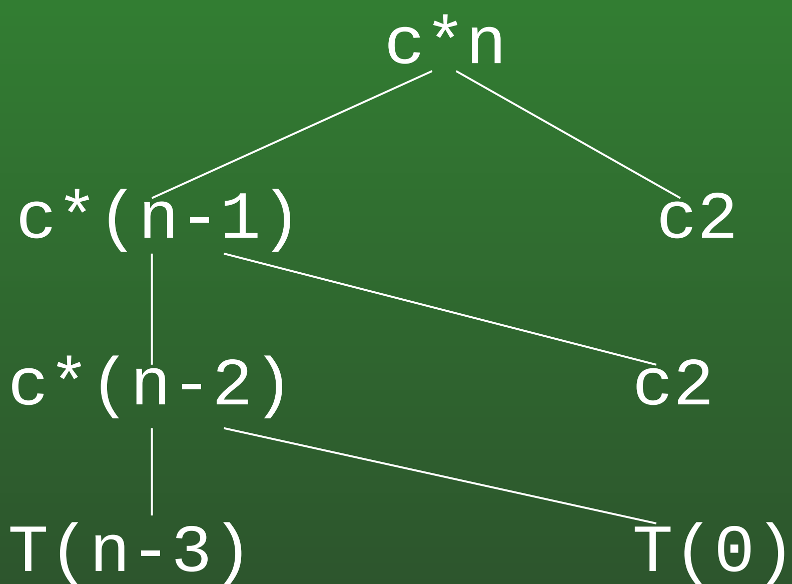
11-30: $\Theta()$ for Quick Sort

```
          c*n
         /    \
T(n-1)           T(0)
```

11-31: $\Theta()$ for Quick Sort



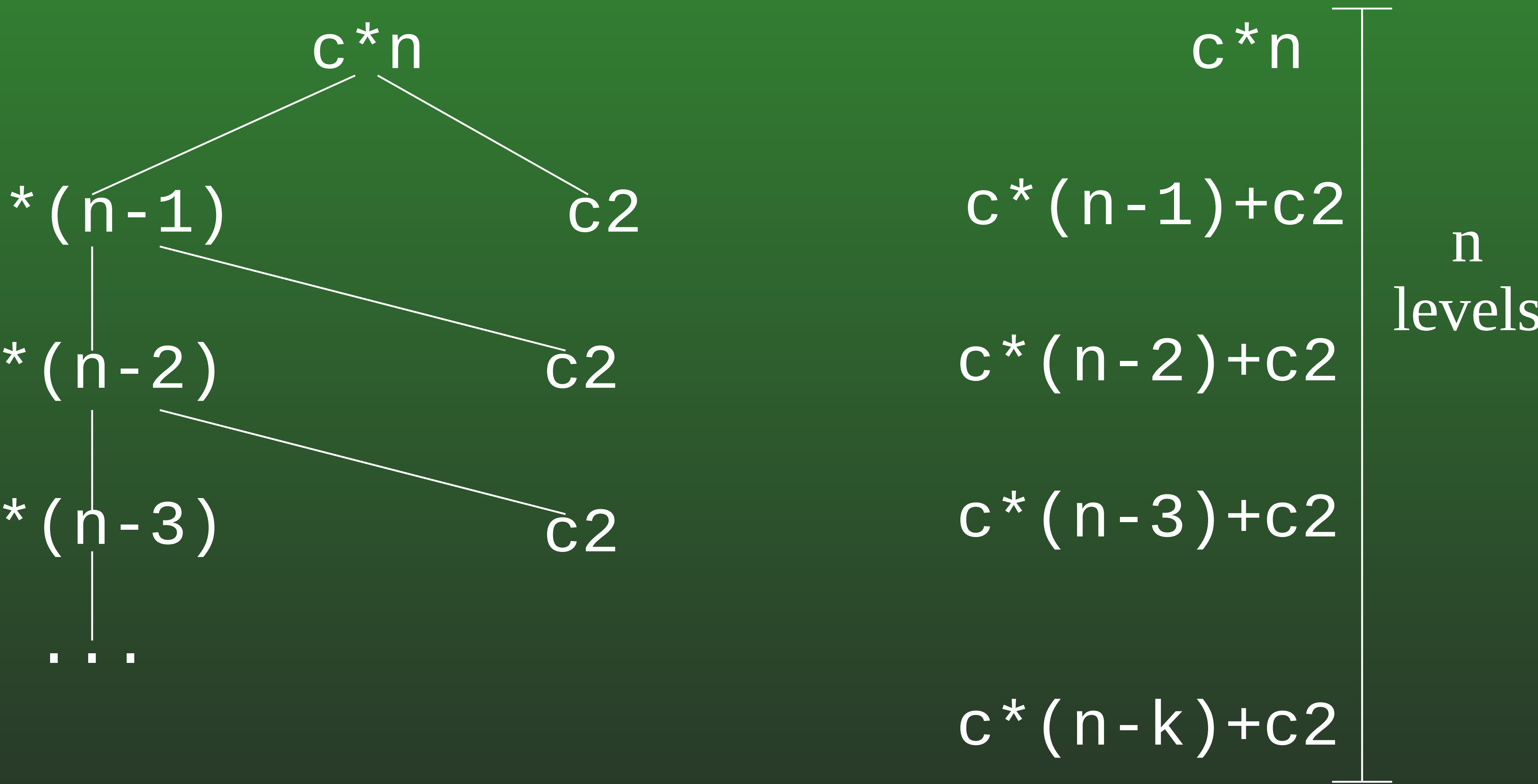
11-32: $\Theta()$ for Quick Sort



11-33: $\Theta()$ for Quick Sort

```
                c*n                                         c*n
              /     \
*(n-1)                 c2                          c*(n-1)+c2
 |     \                                                           n
*(n-2)                 c2                          c*(n-2)+c2   levels
 |     \
*(n-3)                 c2                          c*(n-3)+c2
 |
 ...
                                                   c*(n-k)+c2
```

$\Theta()$ for Quick Sort



Total time = c*n*(n+1)/2 + nc2

$\Theta(n^2)$

11-35: $\Theta()$ for Quick Sort

Best case performance occurs when break list into size $\lfloor (n-1)/2 \rfloor$ and size $\lceil (n-1)/2 \rceil$

$T(0) = c_1$	for some constant c_1
$T(1) = c_2$	for some constant c_2
$T(n) = nc_3 + 2T(n/2)$	for some constant c_3

This is the same as Merge Sort: $\Theta(n \lg n)$

Quick Sort?

If Quicksort is $\Theta(n^2)$ on some lists, why is it called *quick*?

- Most lists give running time of $\Theta(n \lg n)$: The average case running time (assuming all permutations are equall likely) is $\Theta(n \lg n)$
  - We could prove this by finding the running time for each permutation of a list of length n, and averaging them
    - Math required to do this is a little beyond the prerequisites for this class
  - Consider what happens when the list is always partitioned into a list of length $n/9$ and a list of lenth $8n/9$ (recursion tree, on whiteboard)

11-37: *Quick* Sort?

If Quicksort is $\Theta(n^2)$ on some lists, why is it called *quick*?

- Most lists give running time of $\Theta(n \lg n)$
  - Average case running time is $\Theta(n \lg n)$
- Constants are very small
  - Constants don't matter when complexity is different
  - Constants *do* matter when complexity is the same

What lists will cause Quick Sort to have $\Theta(n^2)$ performance?

11-38: Quick Sort - Worst Case

- Quick Sort has worst-case performance when:
  - The list is sorted (or almost sorted)
  - The list is inverse sorted (or almost inverse sorted)
- Many lists we want to sort are almost sorted!
- How can we fix Quick Sort?

11-39: Better Partitions

- Pick the middle element as the pivot
  - Sorted and reverse sorted lists give good performance
- Pick a random element as the pivot
  - No single list always gives bad performance
- Pick the median of 3 elements
  - First, Middle, Last
  - 3 Random Elements

11-40: Improving Quick Sort

- Insertion Sort runs faster than Quick Sort on small lists
  - Why?
- We can combine Quick Sort & Insertion Sort
  - When lists get small, run Insertion Sort instead of a recursive call to Quick Sort
  - When lists get small, stop! After call to Quick Sort, list will be almost sorted – finish the job with a single call to Insertion Sort

11-41: Heap Sort

- Copy the data into a new array (except leave out element at index 0)
- Build a heap out of the new array
- Repeat:
  - Remove the smallest element from the heap, add it to the original array
- Until all elements have been removed from the heap
- The original array is now sorted

Example: 3 1 7 2 5 4

11-42: Heap Sort

- This requires $\Theta(n)$ extra space
- We can modify heapsort so that it does not use extra space
- Build a heap out of the original array, with two differences:
  - Consider element 0 to be the root of the tree
    - for element i, children are at 2*i +1 and 2*i+2, and parent is at $(i-1)/2$
    - (examples)
  - Max-heap instead of a standard min-heap
    - For each subtree, element stored at root $\geq$ element stored in that subtree (instead of $\leq$, as in a standard heap)

Heap Sort

- Build a heap out of the original array, with two differences:
  - Consider element 0 to be the root of the tree
    - for element i, children are at 2*i +1 and 2*i+2, and parent is at $(i-1)/2$
    - (examples)
  - Max-heap instead of a standard min-heap
    - For each subtree, element stored at root $\geq$ element stored in that subtree (instead of $\leq$, as in a standard heap)
- Repeatedly remove the largest element, and insert it in the back of the heap

Example: 3 1 7 2 5 4

$\Theta()$ for Heap Sort

- Building the heap takes time $\Theta(n)$
- Each of the n RemoveMax calls takes time $O(\lg n)$
- Total time: $(n \lg n)$ (also $\Theta(n \lg n)$)

Stability

Sorting Algorithm	Stable?
Insertion Sort	
Selection Sort	
Bubble Sort	
Shell Sort	
Merge Sort	
Quick Sort	
Heap Sort	

Stability

Sorting Algorithm	Stable?
Insertion Sort	Yes
Selection Sort	No
Bubble Sort	Yes
Shell Sort	No
Merge Sort	Yes
Quick Sort	No
Heap Sort	No