Day 10

COMP1006/1406

Summer 2016

M. Jason Hinek
Carleton University

today's agenda

- assignments
  - Only the Project is left!
- Recursion Again
- Efficiency

last time...

recursion... binary trees...

binary trees

A **binary tree** is

- empty (base case), or
- an item and two binary trees (called left and right)

```
public class BTNode{
  String data;     // or int data; etc..
  BTNode left;
  BTNode right;
}
```

We may use a `BinaryTree` class that is a reference to the root of the tree or we can just represent binary trees with a `BTNode`

```
public class BinaryTree{
  BTNode root;
  int size;     // maybe store the size
}
```

binary trees

A binary tree traversal involves two recursive calls (one for each child). The convention is to visit the left child before the right child.

```
public void traverse(BTNode tree){
   if(tree==null) return;

   // do something with tree.data  // preorder traversal

   traverse( tree.root.left );

   // do something with tree.data  // inorder traversal

   traverse( tree.root.right );

   // do something with tree.data  // postorder traversal

}
```

binary trees

Example : count the nodes in a binary tree

```
public int size(BTNode tree){
   /* just use a node for a tree in this example */


   /* base case : empty tree has no nodes */
   if(root==null) return 0;

   /* recursive case : size is 1 plus size of subtrees */
   return 1 + size(root.left) + size(root.right);
}
```

binary trees

Example : sum the data values in a binary tree (assume nodes store ints)

```
public int sumOfTree(BinaryTree tree){
   /* call recursive helper function */
   return sumHelper(tree.root);
}

public static int sumHelper(BTNode root){
   /* base case : empty tree has no values */
   if(root==null) return 0;

   /* recursive case : add current node to sum of subtrees */
   return root.data + sumHelper(root.left) + sumHelper(root.right);
}
```

binary trees

Example : find the max value in a binary tree (assume nodes store ints)

```
public int maxOfTree(BinaryTree tree){
   /* precondition : tree is not null */

   return maxHelper(tree.root, tree.root.data);
}

public static int maxHelper(BTNode root, int big){
   /* precondition : big is the max value seen so far  */

   /* base case : return biggest value seen so far */
   if(root==null) return big;

   /* recursive case  */
   big = Math.max(big, root.data);
   big = Math.max(big, maxHelper(root.left, big));
   big = Math.max(big, maxHelper(root.right, big));

   return big;
}
```

Tail Recursion

There is overhead involved when using recursion.

- each time a function is called a new activation record is pushed to the stack (this costs time and uses up stack space)

Consider the two recursive functions

```
int sum1(LinkedList list){
  if(list.size() == 0){ return 0; }
  return list.first() + sum1(list.rest());
}

int sum2(LinkedList list){
  return sumHelper(list, 0);
}

int sumHelper(LinkedList list, int accumulator){
  if(list.size()==0){ return accumulator; }
  return sumHelper(list.rest(), accumulator+list.first());
}
```

Tail Recursion

The helper function (sumHelper) is an example of **tail recursion**. In tail recursion, the very last operation of the method is simply a recursive call to itself. If the function returns a value then the return value is simply the value returned from the recursive call.

Remember that each time a function is called a new activation record is pushed to the stack (this costs time and uses up stack space). For a function like sumHelper this is a waste of resources.

Some languages (or compilers) can optimize code using tail recursion by only creating a single activation record on the stack and reusing it for each recursive call. This saves time and space. Scheme is a language that guarantees optimized tail recursion. You do not need to worry about running out of stack space when using recursion in Scheme. (why?)

Last Quiz!

Efficiency

how do we know if a program or a method is efficient?

Efficiency

how do we know if a program or a method is efficient?

Efficiency

how do we know if a program or a method is efficient?

Efficiency

how do we know if a program or a method is efficient?

- count basic operations (such as comparisons or swaps)
  - as a function of input size (typically n)
  - length of array, number of nodes in a linked list or tree, size of set
- which input do we consider?
  - worst case
  - average case
  - best case

Efficiency

how do we know if a program or a method is efficient?

- count basic operations (such as comparisons or swaps)
  - as a function of input size (typically *n*)
  - length of array, number of nodes in a linked list or tree, size of set
- which input do we consider?
  - worst case ✓✓ (typical)
  - average case ✓ (sometimes hard to determine)
  - best case ✗(✓) (winning the lottery)

Efficiency

how do we know if a program or a method is efficient?

- count basic operations (such as comparisons or swaps)
  - as a function of input size (typically n)
  - length of array, number of nodes in a linked list or tree, size of set
- which input do we consider?
  - worst case ✓✓ (typical)
  - average case ✓ (sometimes hard to determine)
  - best case ✗(✓) (winning the lottery)
- use big-O notation for runtime
  - if number of operations is $3n^3 + 8n^2 - 1000n + 3 \rightarrow \boxed{O(n^3)}$
  - if number of operations is $\frac{n^2}{100000} + 800000n \rightarrow \boxed{O(n^2)}$
  - ignore lower "order" terms and constants

Efficiency

there are several common complexities (runtimes)

- constant $\sim c$ — $O(1)$
- logarithmic $\sim c\log(n)$ — $O(\log n)$
- linear $\sim cn$ — $O(n)$
- linearithmic $\sim cn\log(n)$ — $O(n\log n)$
- quadratic $\sim cn^2$ — $O(n^2)$
- cubic $\sim cn^3$ — $O(n^3)$
- exponential $\sim c2^n$ — $O(2^n)$

Efficiency

we need to be very **careful** how we interpret big-O runtimes.

- big-O is an **asymptotic** result
  - it only holds for big input values
- in practice the ignored terms (constants and lower order) can matter
  - if algorithm A has runtime $O(n^2)$ and algorithm B has runtime $O(n)$ which is faster?
- algorithms with the same big-O may behave differently
  - n^3 and $1000000n^3 + 10000000000000n^2$ are both $O(n^3)$
- it does give us valuable information about the runtime behaviour
  - it tells us the **growth rate** of the runtime

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c		
logarithmic	$c\log(n)$		
linear	cn		
linearithmic	$cn\log(n)$		
quadratic	cn^2		
cubic	cn^3		
exponential	$c2^n$		

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$		
linear	cn		
linearithmic	$cn\log(n)$		
quadratic	cn^2		
cubic	cn^3		
exponential	$c2^n$		

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn		
linearithmic	$cn\log(n)$		
quadratic	cn^2		
cubic	cn^3		
exponential	$c2^n$		

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$		
quadratic	cn^2		
cubic	cn^3		
exponential	$c2^n$		

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2		
cubic	cn^3		
exponential	$c2^n$		

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2	$4cn^2$	$4T(n)$
cubic	cn^3		
exponential	$c2^n$		

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2	$4cn^2$	$4T(n)$
cubic	cn^3	$8cn^3$	$8T(n)$
exponential	$c2^n$		

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2	$4cn^2$	$4T(n)$
cubic	cn^3	$8cn^3$	$8T(n)$
exponential	$c2^n$	$c2^{2n}$	$2^nT(n)$

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2	$4cn^2$	$4T(n)$
cubic	cn^3	$8cn^3$	$8T(n)$
exponential	$c2^n$	$c2^{2n}$	$2^nT(n)$

what happens if we increase the input size by 1?

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2	$4cn^2$	$4T(n)$
cubic	cn^3	$8cn^3$	$8T(n)$
exponential	$c2^n$	$c2^{2n}$	$2^nT(n)$

what happens if we increase the input size by 1?

	$T(n)$	$T(n+1)$	T(n+1)
linear	cn	cn	$T(n)$
quadratic	cn^2	cn^2	$T(n)$

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2	$4cn^2$	$4T(n)$
cubic	cn^3	$8cn^3$	$8T(n)$
exponential	$c2^n$	$c2^{2n}$	$2^nT(n)$

what happens if we increase the input size by 1?

	$T(n)$	$T(n+1)$	T(n+1)
linear	cn	cn	$T(n)$
quadratic	cn^2	cn^2	$T(n)$
exponential	$c2^n$		

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2	$4cn^2$	$4T(n)$
cubic	cn^3	$8cn^3$	$8T(n)$
exponential	$c2^n$	$c2^{2n}$	$2^nT(n)$

what happens if we increase the input size by 1?

	$T(n)$	$T(n+1)$	T(n+1)
linear	cn	cn	$T(n)$
quadratic	cn^2	cn^2	$T(n)$
exponential	$c2^n$	$c2^{n+1}$	

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)
constant	c	c	$T(n)$
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$
linear	cn	$2cn$	$2T(n)$
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$
quadratic	cn^2	$4cn^2$	$4T(n)$
cubic	cn^3	$8cn^3$	$8T(n)$
exponential	$c2^n$	$c2^{2n}$	$2^nT(n)$

what happens if we increase the input size by 1?

	$T(n)$	$T(n+1)$	T(n+1)
linear	cn	cn	$T(n)$
quadratic	cn^2	cn^2	$T(n)$
exponential	$c2^n$	$c2^{n+1}$	$2T(n)$

Efficiency

what happens is we double the input size? (ignoring lower order terms)

	$T(n)$	$T(2n)$	T(2n)	
constant	c	c	$T(n)$	
logarithmic	$c\log(n)$	$c\log(n)$	$T(n)$	
linear	cn	$2cn$	$2T(n)$	
linearithmic	$cn\log(n)$	$2cn\log(n)$	$2T(n)$	
quadratic	cn^2	$4cn^2$	$4T(n)$	
cubic	cn^3	$8cn^3$	$8T(n)$	
exponential	$c\kappa^n$	$c\kappa^{2n}$	$\kappa^n T(n)$	any $\kappa > 1$

what happens if we increase the input size by 1?

	$T(n)$	$T(n+1)$	T(n+1)
linear	cn	cn	$T(n)$
quadratic	cn^2	cn^2	$T(n)$
exponential	$c\kappa^n$	$c\kappa^{n+1}$	$\kappa T(n)$

Searching

Searching is a **fundamental** operation in computer science

is there a difference if the data is unordered or ordered?

Searching

Searching is a **fundamental** operation in computer science

is there a difference if the data is unordered or ordered?

- unordered list
  - when we find find the target we find the target
  - how do we know if target is not in the list?
  - we have to look at every element in the list... $O(n)$
- ordered list
  - can we use this extra information to help us?
  - data stored in an array (ArrayList)
    - binary search (eliminate 1/2 of search space with each comparison)
    - $O(\log(n))$
  - data stored in a linked list?
    - we have to walk through the list to look for the element
    - $O(n)$
- how we store our data is important! (COMP2402)

Arrays vs Linked Lists

let's compare arrays and linked lists

	array	linked list
search an unsorted list		
search an sorted list		
add/remove at front of list		
add/remove at back of list		
add/remove from arbitrary position		
access element in arbitrary position		

So which implementation of a list is better?

Arrays vs Linked Lists

let's compare arrays and linked lists

	array	linked list
search an unsorted list	$O(n)$	$O(n)$
search an sorted list	$O(\log n)$	$O(n)$
add/remove at front of list	$O(n)$	$O(1)$
add/remove at back of list	$O(1)$	$O(1)$
add/remove from arbitrary position	$O(n)$	$O(n)$
access element in arbitrary position	$O(1)$	$O(n)$

So which implementation of a list is better?

Sorting

sorting is another **fundamental** operation in computer science

you should have seen some sorting algorithms in COMP1004/1405 (bubble sort, insertion sort, selection sort). These were all quadratric sorting algorithms $O(n^2)$

let's try to do better. But instead of jumping into sorting let's look at a different problem and apply divide and conquer:

suppose we have two lists of numbers with n numbers in total

Sorting

sorting is another **fundamental** operation in computer science

you should have seen some sorting algorithms in COMP1004/1405 (bubble sort, insertion sort, selection sort). These were all quadratic sorting algorithms $O(n^2)$

let's try to do better. But instead of jumping into sorting let's look at a different problem and apply divide and conquer:

suppose we have two lists of numbers with n numbers in total

- how efficiently can we create a new list with all the numbers in sorted order?
  - does it matter that we have two lists? no
- what if both lists were each sorted lists?
  - a-ha!
  - $O(n)$
- we have the `merge` algorithm
  - $O(n)$
    when merging two lists whose lengths add to n

Recursive Sorting I

the merge algorithm takes two sorted lists and creates a single sorted list with all the elements

```
merge(List a, List b):
   list = new empty list
   while a is not empty OR b is not empty
      compare the first elements of a and b
      remove the greater from its list and add to list
   return list
```

Suppose your favourite sorting algorithm takes cn^2 time to sort n elements and merge takes n time.

- split the list in two
- sort each in time $c(n/2)^2 = \frac{1}{4}cn^2$
- merge the two sorted lists in linear time $c'n$
- total time is $\frac{1}{2}cn^2 + n$ (still $O(n^2)$ but better!)

Recursive Sorting I

the `mergesort` algorithm recursively splits up the list until the lists are small enough to sort (base case) and then merges the results. (assume $n = 2^\ell$)

- divide list of n numbers into n lists of one number (base case)
- call `merge` on pairs of single number lists
  (giving 2-element sorted lists)
- call `merge` on pairs of 2-element sorted lists
  (giving 4-element sorted lists)
- ...
- call `merge` on the two $n/2$-element sorted lists giving a single n-element sorted list

Recursive Sorting I

`mergesort` (assume $n = 2^\ell$)

```
mergesort(list[1..n]):
  if size of list is 1 then return list

  left = mergesort( list.sublist(1,n/2) )
  right = mergesort( list.sublist(n/2+1, n) )

  return merge(left,right)
```

Recursive Sorting I

the `mergesort` (assume $n = 2^\ell$)

```
mergesort(list[1..n]):
  if size of list is 1 then return list

  left = mergesort( list.sublist(1,n/2) )
  right = mergesort( list.sublist(n/2+1, n) )

  return merge(left,right)
```

what is the runtime of this algorithm?

Recursive Sorting I

the `mergesort` (assume $n = 2^\ell$)

```
mergesort(list[1..n]):
  if size of list is 1 then return list

  left = mergesort( list.sublist(1,n/2) )
  right = mergesort( list.sublist(n/2+1, n) )

  return merge(left,right)
```

what is the runtime of this algorithm?

$$T(n) = \begin{cases} c & n = 1 \\ T(n/2) + T(n/2) + Merge(n) & n > 1 \end{cases}$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^{\ell}$)

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$T(n) \\ = T(n/2) + T(n/2) + M(n)$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n)
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n)
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n) \\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n)
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n) \\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n) \\
&\quad = \ldots
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n) \\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n) \\
&\quad = \ldots \\
&\quad = 2^\ell T(1) + 2^{\ell-1} M(n/2^{\ell-1}) + \cdots + 4M(n/4) + 2M(n/2) + M(n)
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n) \\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n) \\
&\quad = \ldots \\
&\quad = 2^\ell T(1) + 2^{\ell-1} M(n/2^{\ell-1}) + \cdots + 4M(n/4) + 2M(n/2) + M(n) \\
&\quad \downarrow \;\; T(1) \sim c, \; M(x) \sim kx
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n) \\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n) \\
&\quad = \ldots \\
&\quad = 2^\ell T(1) + 2^{\ell-1} M(n/2^{\ell-1}) + \cdots + 4M(n/4) + 2M(n/2) + M(n) \\
&\quad \downarrow \;\; T(1) \sim c, \; M(x) \sim kx \\
&\quad = 2^\ell c + 2^{\ell-1} k(n/2^{\ell-1}) + \cdots + 4k(n/4) + 2k(n/2) + kn
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n)\\
&\quad = T(n/2) + T(n/2) + M(n)\\
&\quad = 2T(n/2) + M(n)\\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n)\\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n)\\
&\quad = \ldots\\
&\quad = 2^\ell T(1) + 2^{\ell-1} M(n/2^{\ell-1}) + \cdots + 4M(n/4) + 2M(n/2) + M(n)\\
&\quad \downarrow \;\; T(1) \sim c,\; M(x) \sim kx\\
&\quad = 2^\ell c + 2^{\ell-1} k(n/2^{\ell-1}) + \cdots + 4k(n/4) + 2k(n/2) + kn\\
&\quad = 2^\ell c + \ell kn
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n) \\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n) \\
&\quad = \dots \\
&\quad = 2^\ell T(1) + 2^{\ell-1} M(n/2^{\ell-1}) + \cdots + 4M(n/4) + 2M(n/2) + M(n) \\
&\quad \downarrow \;\; T(1) \sim c, \; M(x) \sim kx \\
&\quad = 2^\ell c + 2^{\ell-1} k(n/2^{\ell-1}) + \cdots + 4k(n/4) + 2k(n/2) + kn \\
&\quad = 2^\ell c + \ell kn \\
&\quad = 2^\ell c + \log_2(2^\ell) kn
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
&T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n) \\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n) \\
&\quad = \dots \\
&\quad = 2^\ell T(1) + 2^{\ell-1} M(n/2^{\ell-1}) + \cdots + 4M(n/4) + 2M(n/2) + M(n) \\
&\quad \downarrow \;\; T(1) \sim c,\; M(x) \sim kx \\
&\quad = 2^\ell c + 2^{\ell-1} k(n/2^{\ell-1}) + \cdots + 4k(n/4) + 2k(n/2) + kn \\
&\quad = 2^\ell c + \ell kn \\
&\quad = 2^\ell c + \log_2(2^\ell) kn \\
&\quad = nc + \log_2(n) kn
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

$$
\begin{aligned}
& T(n) \\
&\quad = T(n/2) + T(n/2) + M(n) \\
&\quad = 2T(n/2) + M(n) \\
&\quad = 2\underbrace{(2T(n/4) + M(n/2))}_{T(n/2)} + M(n) \\
&\quad = 2(2\underbrace{(2T(n/8) + M(n/4))}_{T(n/4)} + M(n/2)) + M(n) \\
&\quad = \ldots \\
&\quad = 2^\ell T(1) + 2^{\ell-1} M(n/2^{\ell-1}) + \cdots + 4M(n/4) + 2M(n/2) + M(n) \\
&\quad \downarrow \;\; T(1) \sim c,\; M(x) \sim kx \\
&\quad = 2^\ell c + 2^{\ell-1} k(n/2^{\ell-1}) + \cdots + 4k(n/4) + 2k(n/2) + kn \\
&\quad = 2^\ell c + \ell kn \\
&\quad = 2^\ell c + \log_2(2^\ell) kn \\
&\quad = nc + \log_2(n) kn \\
&\quad = O(n \log n)
\end{aligned}
$$

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

that was a bit traumatic...

Recursive Sorting I

what is the runtime of the `mergesort` (suppose $n = 2^\ell$)

that was a bit traumatic...

let's draw a picture instead

Recursive Sorting II

the mergesort was efficient because merging two sorted lists is efficient.

Recursive Sorting II

the mergesort was efficient because merging two sorted lists is efficient.

let's look at another problem on lists

Recursive Sorting II

the mergesort was efficient because merging two sorted lists is efficient.

let's look at another problem on lists

problem: given a list ℓ and one element in the list α, can you partition the list so that all elements before α are less than α and all elements after α are greater than or equal to α?

Recursive Sorting II

the mergesort was efficient because merging two sorted lists is efficient.

let's look at another problem on lists

problem: given a list ℓ and one element in the list α, can you partition the list so that all elements before α are less than α and all elements after α are greater than or equal to α?

12	3	-12	223	62	17	99	78	82	101	11

Recursive Sorting II

the mergesort was efficient because merging two sorted lists is efficient.

let's look at another problem on lists

problem: given a list ℓ and one element in the list α, can you partition the list so that all elements before α are less than α and all elements after α are greater than or equal to α?

12	3	-12	223	62	17	99	78	82	101	11

$\downarrow$ $\alpha = 78$

Recursive Sorting II

the mergesort was efficient because merging two sorted lists is efficient.

let's look at another problem on lists

problem: given a list ℓ and one element in the list α, can you partition the list so that all elements before α are less than α and all elements after α are greater than or equal to α?

12	3	-12	223	62	17	99	78	82	101	11

$\downarrow \quad \alpha = 78$

12	3	-12	62	17	11	78	223	99	82	101

Recursive Sorting II

the **quicksort** method keeps partitioning lists and then concatenating them back together

Recursive Sorting II

the **quicksort** method keeps partitioning lists and then concatenating them back together

```
quicksort(list)
    if size of list is 1 then return list

    q := partition
    left := elements in list < q
    right := elements in list >= q

    return quicksort(left) + {q} || quicksort(right)
```