CS280- Data Structures

Introductory Sorting Part 2

Recursive Algorithms

Terminologies

- Recursive definition: A definition in which something is defined in terms of smaller versions of itself.
- Base case : The case for which the solution can be stated non-recursively.
- General case (recursive case) : The case for which the solution is expressed in terms of a smaller version of itself.

Terminologies

- Recursive algorithm : A solution that is expressed in terms of
  - a base case
  - a recursive case

- Recursive call : A function call in which the function being called is the same as the one making the call.

- Infinite recursion : The situation in which a function calls itself over and over endlessly.

Recursive Algorithms

- A recursive algorithm is simply one that is defined in terms of itself.
- E.g.:
  - Finding the length of the string
  - Binary search

Recursive Algorithms

```
size_t length(char* s){
if(*s==0)
    return 0;
return 1+length(++s);
}
```

base cases

recursive calls

Recursive Algorithms

```
int binarysearch(int a[], int x, int low, int
   high){
    int mid = (low + high)/2;
    if (low > high)
      return -1;
    else if (a[mid] == x)
      return mid;
    else if (a[mid] < x)
      return bsearch(a, x, mid+1, high);
    else
      return bsearch(a, x, low, mid-1);
}
```

base cases

recursive calls

Terminologies

- Divide and conquer:

- Original problem → **split** into smaller sub-problem
- Subproblem solutions → **combine** to be the solution to the original problem

Reasons for creating sub-problems:

Because the sub problems are …

Reasons for creating sub-problems:

The sub problems are

– smaller or simpler than the original problems

Reasons for creating sub-problems:

- The sub problems are
- smaller or simpler than the original problems
- has an immediate solution, or can be solved by further recursion

Terminologies

- Recursive functions can call themselves either directly or indirectly:
  - Direct - *FunctionA* calls *FunctionA*
  - Indirect
    - *FunctionA* calls *FunctionB, FunctionB* calls *FunctionA*
    - *FunctionA* calls *FunctionB, FunctionB* calls *FunctionC, FunctionC* calls *FunctionA*.

Recursion v.s. Iteration

- Recursion is very much like iteration (looping).
  - In recursion you make a function call.
  - In iteration you jump to the top of a loop
- Anything you can do iteratively, you can do recursively!

Recursion v.s. Iteration

- Counting down from 5 iteratively

Version 1

```
void PrintDown1(void){
  for (int i=5; i>0; --i)
        cout << i << endl;
}
```

Version 2

```
void PrintDown2(void){
  int i = 5;
  while (i > 0)
    cout << i-- << endl;
}
```

Recursion v.s. Iteration

- Counting down from 5 recursively

Version 1

```
int Value = 5;

void PrintDown1(void){
  if (Value < 1)
    return;
  else{
    cout << Value-- << endl;
PrintDown1();
  }
}
```

Version 2

```
int Value = 5;

void PrintDown2(void){
  if (Value > 0){
    cout << Value-- << endl;
   PrintDown2();
  }
}
```

Recursion v.s. Iteration

- Counting down from 5 recursively (using parameters) (Better)

Version 1

```
void PrintDown1(int Value){
  if (Value < 1)
    return;
  else{
    cout << Value << endl;
    PrintDown1(Value - 1);
  }
}
```

Version 2

```
void PrintDown2(int Value){
  if (Value > 0){
    cout << Value << endl;
    PrintDown2(Value - 1);
  }
}
```

Recursion

- Q: Is the recursive version usually faster?

Recursion

- Q: Is the recursive version usually faster?

  A: No -- it's usually **slower**

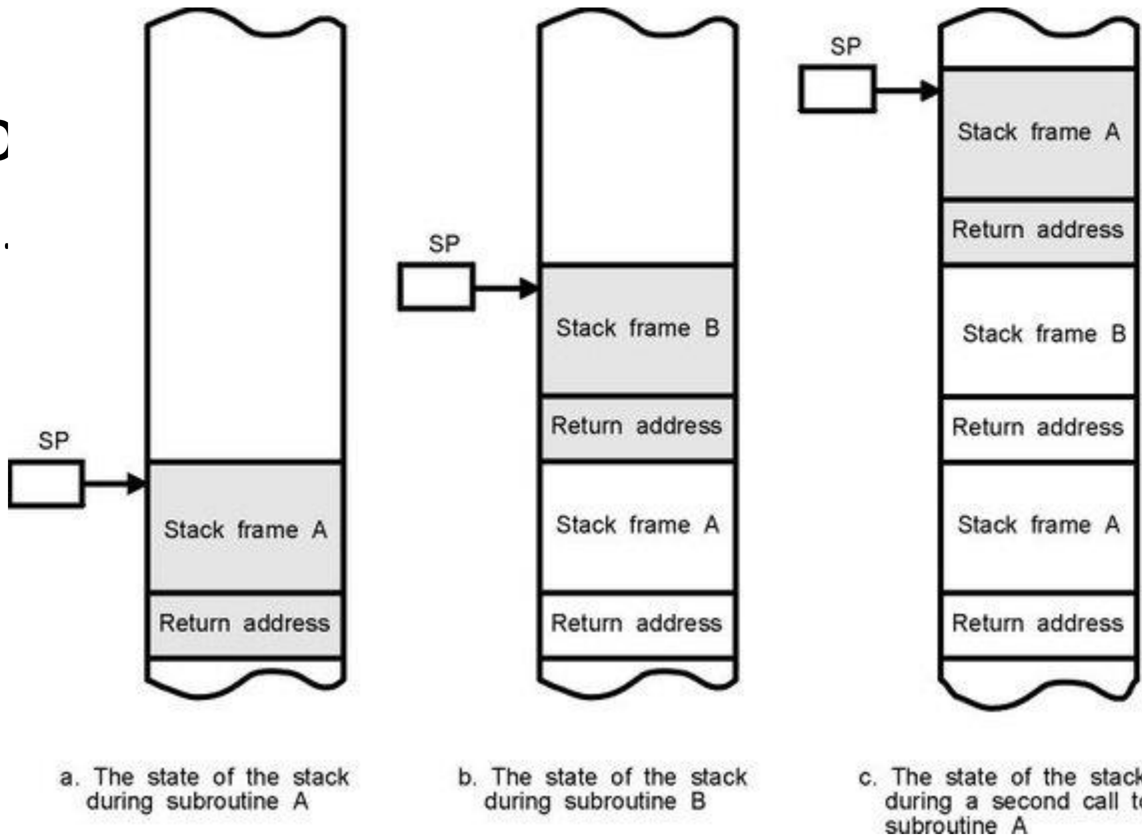
  due to the overhead of maintaining the stack

Recursion

- Q: Is the recursive version usually faster?

  A: No

  due … he stack



Recursion

- Q: Does the recursive version usually use less memory?

Recursion

- Q: Does the recursive version usually use less memory?

  A: No -- it usually uses **more memory** (for the stack).

Recursion

- Q: Then **why** use recursion??

Recursion

- Q: Then **why** use recursion??
-

  A: Sometimes it is much **simpler to write** the recursive version

  (but we'll need to wait until we've discussed **trees** to see really good examples...)

Sorting Algorithms using recursions

- Merge sort
- Quick sort

Sorting Algorithms

- Merge sort
- Quick sort

Merge Sort

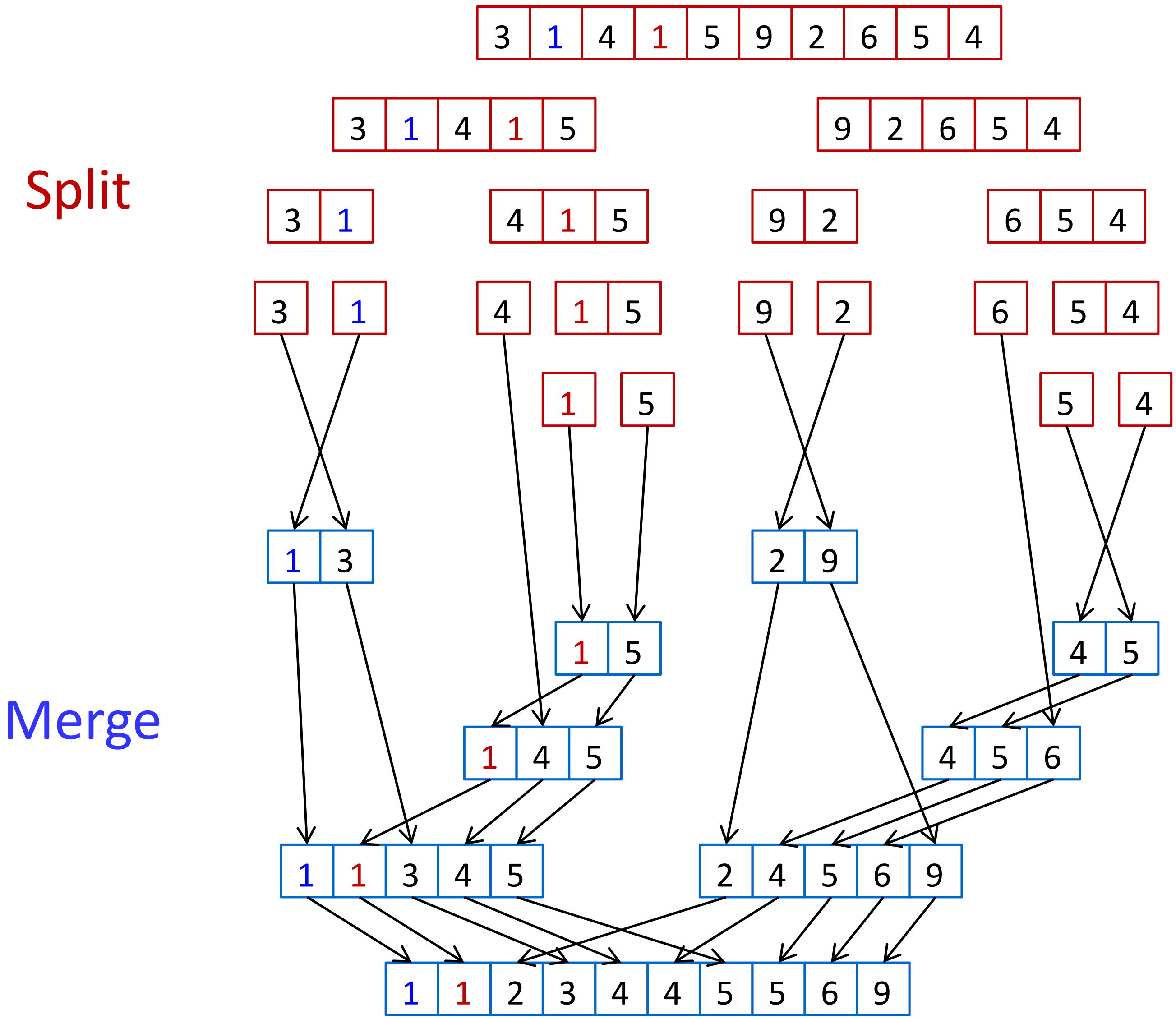
- Main idea: Divide and merge each sub-sequence in order.

1. Recursively divide sequence until single elements
2. Merge them back together in order.

Merge Sort Example

- A[9]={7, 4, 1, 3, 8, 6, 5, 9, 2}
- objective: A’[9]={1, 2, 3, 4, 5, 6, 7, 8, 9}

Merge Sort Example (Duplicates)



Merge Sort

```
void DoMergeSort(int a[], int left, int right){
   if (left < right){
        unsigned const middle = (left+right)/2;
        DoMergeSort(a,left,middle);
        DoMergeSort(a,middle+1,right);
        Merge(a,left,middle,right);
   }
}
```

Merge Sort: Merge function

```
void Merge(int array[], int left, int middle, int right){
   unsigned i = left; // counter for the temp array

   unsigned j = left; // counter for left array
   unsigned k = middle + 1; // counter for right array
   int* temp = new int [right+1];
   while (j<=middle && k <=right)
     if (array[j] <= array[k])
         temp[i++] = array[j++];
     else
         temp[i++] = array[k++];
   while (j <= middle)
         temp[i++] = array[j++];
   while (k <= right)
         temp[i++] = array[k++];
   for (i=left; i <= right; ++i)
       array[i] = temp[i];
   delete [] temp;
}
```

Complexity of Merge Sort

- Complexity Analyses of `Merge()`:
  - Best/Worse/Average case $O(n)$
- Complexity Analyses of `MergeSort()`:
  - Best/Worse/Average case $O(n\log n)$

Sorting Algorithms

- Merge sort
- Quick sort
- Lower bounds for sorting

Quick Sort

- Main idea: Divide and sort each sub-sequence based on a pivoted value recursively.

1. Select a pivot p element from A.
2. Partition the remaining elements in 2 parts L and G:
   a) For each $\mathrm{s} \in \mathrm{L}, \mathrm{s} \leq \mathrm{p}$
   b) For each $\mathrm{s} \in \mathrm{G}, \mathrm{s} > \mathrm{p}$
3. Recursively quicksort the unsorted L & G.

Quick Sort

- [P, [SSSSSS], [LLLLL], C, [RRRRRR]]
- (P=pivot, S=smaller, L=larger, C=current, R=remaining)
- If C ≥ P, then leave it at the same place
- If C < P, then swap it with the first L element
- At the end swap the pivot with the last S element

Quick Sort Example

- A[8]={4, 7, 1, 3, 8, 6, 5, 2}
- (Next item: 7; > pivot, so leave it as it is.)
- 4, 7, 1, 3, 8, 6, 5, 2
- (Next item: 1; < pivot, so swap with 1st large element)
- 4, 1, 7, 3, 8, 6, 5, 2
- (Next item: 3; < pivot, so swap with 1st large element)
- 4, 1, 3, 7, 8, 6, 5, 2
- (Next item: 8; > pivot, so leave it as it is)
- 4, 1, 3, 7, 8, 6, 5, 2
- (Next item: 6; > pivot, so leave it as it is)
- 4, 1, 3, 7, 8, 6, 5, 2
- (Next item: 5; > pivot, so leave it as it is)
- 4, 1, 3, 7, 8, 6, 5, 2
- (Next item: 2; < pivot , so swap with 1st large element)
- 4, 1, 3, 2, 8, 6, 5, 7
- End of list. Swap the pivot element with last smaller element.
- 2, 1, 3, 4, 8, 6, 5, 7

Quick Sort

```
 void QuickSort(int a[], int left, int right){
     if(left < right){
         int i = Partition(a, left, right);
         QuickSort(a,left, i-1);
         QuickSort(a,i+1, right);
     }
 }
unsigned Partition(int a[], int i, int j){
   int p=a[i];
   int h=i; // the position of the first larger element
   for(int k=i+1;k<=j;++k){
     if(a[k]<p){
        ++h;
        Swap(a[k],a[h]);
     }
// else: don’t do anything, move ahead, keep the item as it is
    }

   Swap(a[h],a[i]);
   return h;
}
```

Complexity of Quick Sort

- Worst Case:
  - Array is already sorted.
  - $(n - 1) + (n - 2) + (n - 3) + . . . + 1 = O(n^2)$.

- Best Case:
  - Each round divides the two parts into nearly equal size.
  - Gives complexity $O(n \log_2 n)$.

Quick Sort Properties

- Advantages
  - Straightforward recursion
  - In-place sorting
- Drawbacks
  - Worst case $O(n^2)$
    - Pivot might always be max/min

(But simple enhancements resolve these…)

Randomized Quicksort

```
void RandomQuickSort(int a[], int left, int right){
    if(left < right){
        int i = RandomPartition(a, left, right);
        RandomQuickSort(a,left, i-1);
        RandomQuickSort(a,i+1, right);
    }
}
unsigned RandomPartition(int a[], int i, int j){
   int r = rand() % (j-i)+i+1;
   Swap(a[i],a[r]);
   int p=a[i];
   int h=i;
   for(int k=i+1;k<=j;++k)
     if(a[k]<p){
       ++h;
       Swap(a[k],a[h]);
     }
   Swap(a[h],a[i]);
   return h;
}
```

Summary

- Merge sort
- Quick sort