a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

Sorting

Iterative sorting
Recursive sorting
Recursive analysis

Categorising sorting algorithms

- Computational complexity
  - Average case behaviour
  - Worst/best case behaviour
    - Why do we care about these?

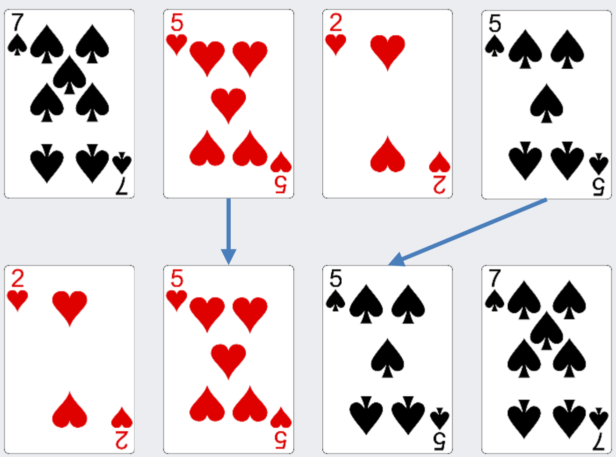
- Memory usage
  - How much *extra* memory is used (outside of the original array)?

- Stability
  - A *stable* sorting algorithm maintains the relative order of records with equal keys

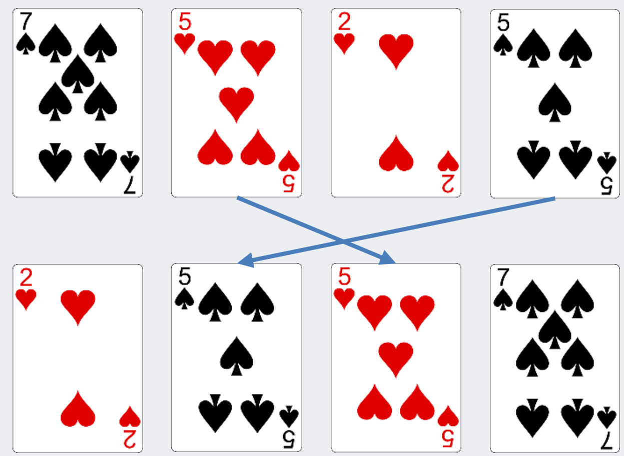
Stability

Definition

- A *stable* sorting algorithm maintains the relative order of records with equal keys
  - for two records x and y with equal keys, if x appears to the left of y in the unsorted input, x still appears to the left of y in the sorted output



Stable



Not stable

Selection sort

- Selection sort is a simple sorting algorithm that repeatedly finds the smallest item
  - The array is divided into a sorted part and an unsorted part
- Repeatedly swap the first unsorted item with the smallest unsorted item
  - Starting with the element with index 0, and
  - Ending with last but one element (index $n - 1$)

Selection sort

23	41	33	81	**7**	19	11	45	Find smallest unsorted item: 7 comparisons
7	41	33	81	23	19	**11**	45	Find smallest unsorted item: 6 comparisons
7	11	33	81	23	**19**	41	45	Find smallest unsorted item: 5 comparisons

ordered

unordered

7	11	19	81	**23**	33	41	45	Find smallest unsorted item: 4 comparisons
7	11	19	23	81	**33**	41	45	Find smallest unsorted item: 3 comparisons
7	11	19	23	33	81	**41**	45	Find smallest unsorted item: 2 comparisons
7	11	19	23	33	41	81	**45**	Find smallest unsorted item: 1 comparison
7	11	19	23	33	41	45	81	Sorted

Number of comparison operations

Selection sort

$$\text{Comparisons} = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Unsorted elements	Comparisons
n	$n-1$
$n-1$	$n-2$
$\dots$	$\dots$
3	2
2	1
1	0
	$n(n-1)/2$

Selection sort implementation

```
void selectionSort(int arr[], int size)
{
  int i; // next index to be set to minimum
  int min_pos; // index of minimum element
  for (i = 0; i < size-1; i++) {
    min_pos = minPosition(arr, i, size-1)
    if (min_pos != i)
      swap(&arr[min_pos], &arr[i]);
  }
}
```

```
int minPosition(int arr[], int start, int end)
{
  int min_pos = start;
  int j;
  for (j = start + 1; j <= end; j++) {
    if (arr[j] < arr[min_pos])
      min_pos = j;
  }
  return min_pos;
}
```

Stability of selection sort

- Is Selection sort stable?

9	4	9	2

2	4	9	9

  - No, but it can be made stable (at the expense of performing many more swaps)

Selection sort summary

- In broad terms and ignoring the actual number of executable statements, selection sort $\Theta(n^2)$
  - Makes $n * (n-1)/2$ comparisons, regardless of the original order of the input
  - Performs $n-1$ swaps: # of write operations is $\Theta(n)$
- Neither of these operations are substantially affected by the organization of the input
  - Selection sort is thus a good choice in systems where write operations are expensive (e.g. in-place sorting on flash memory or an external hard drive)

Name	**Best**	**Average**	**Worst**	**Stable**	**Memory**
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	challenging	$\Theta(1)$

lab 5 quiz content ends here.

Insertion sort

- Another simple sorting algorithm
  - Divides array into sorted and unsorted parts
- The sorted part of the array is expanded one element at a time
  - Find the correct place in the sorted part to place the 1st element of the unsorted part
    - by doing a (backwards) linear search from the back of the sorted portion
  - Move the elements after the insertion point up one position to make space

Insertion sort

First element is already "sorted"

23	41	33	81	7	19	11	45	Locate position for 41 – 1 comparison
23	41	33	81	7	19	11	45	Locate position for 33 – 2 comparisons
23	33	41	81	7	19	11	45	Locate position for 81 – 1 comparison
23	33	41	81	7	19	11	45	Locate position for 7 – 4 comparisons
7	23	33	41	81	19	11	45	Locate position for 19 – 5 comparisons
7	19	23	33	41	81	11	45	Locate position for 11 – 6 comparisons
7	11	19	23	33	41	81	45	Locate position for 45 – 2 comparisons
7	11	19	23	33	41	45	81	Sorted

Insertion sort implementation

outer loop:
process one unordered item

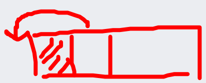
```
void insertionSort(int arr[], int size)
{
  int i, temp, position;
  for (i = 1; i < size; i++)
  {
    temp = arr[i];
    position = i;
    // Shuffle up all sorted items > arr[i]
    while (position > 0 && arr[position - 1] > temp)
    {
      arr[position] = arr[position - 1];
      position--;
    }
    // Insert the current item
    arr[position] = temp;
  }
}
```

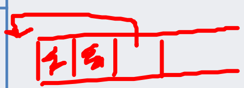
first unordered element

open space for insertion

Insertion sort cost

4 3 2 1
3 4 2 1
2 3 4 1
1 2 3 4





Sorted Elements	Worst-case Search	Worst-case Shuffle
~~0~~	~~0~~	~~0~~
1	1	1
2	2	2
…	…	…
$n-1$	$n-1$	$n-1$
	$n(n-1)/2$	$n(n-1)/2$

$O(n^2)$

worst case: array is in decreasing order

Insertion sort best case

- The efficiency of insertion sort *is* affected by the state of the array to be sorted
- In the best case the array is already completely sorted!
  - No movement of array elements is required
  - Requires n -1 comparisons

1 comparison × (n−1) elements
O(n)

1 2 3 4
1 2 3 4
1 2 3 4

suppose array is completely ordered except for 10 elements → as far out of place as possible

total cost of comparisons : 1 comparison × (n−9) elements
+ n−10
+ n−9
+ n−8
+ ...
n−1

each is still < n × 10

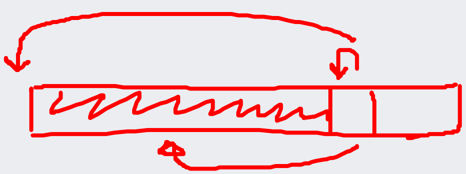
maximum number of elements out of place

Insertion sort worst case

- In the worst case the array is in reverse order
- Every item has to be moved all the way to the front of the array
  - The outer loop runs $n - 1$ times
    - In the first iteration, one comparison and move
    - In the last iteration, $n - 1$ comparisons and moves
    - On average, $n/2$ comparisons and moves
  - For a total of $n \ast (n - 1) / 2$ comparisons and moves

Insertion sort average case

- What is the average case cost?
  - Is it closer to the best case? $O(n)$
  - Or the worst case? $O(n^2)$
- If *random* data is sorted, insertion sort is usually closer to the worst case
  - Around $n \, * \, (n-1) \, / \, 4$ comparisons $O(n^2)$



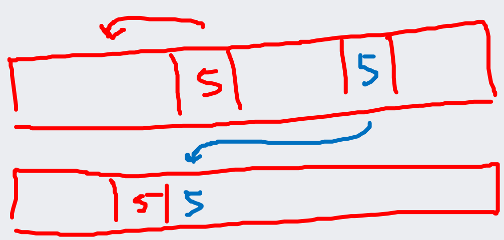
worst case	avg.
1	1/2
2	2/2
3	3/2
4	4/2
⋮	⋮
n-1	(n-1)/2
$\sum_{i=1}^{n-1} i$	$\frac{1}{2} \cdot \sum_{i=1}^{n-1} i$

When is insertion sort used?

- Insertion sort is a good choice when the data are nearly sorted (only a few elements out of place), or when the problem size is small (because it has low overhead)

constant upper bound

Name	Best	Average	Worst	Stable	Memory
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$



Merge sort

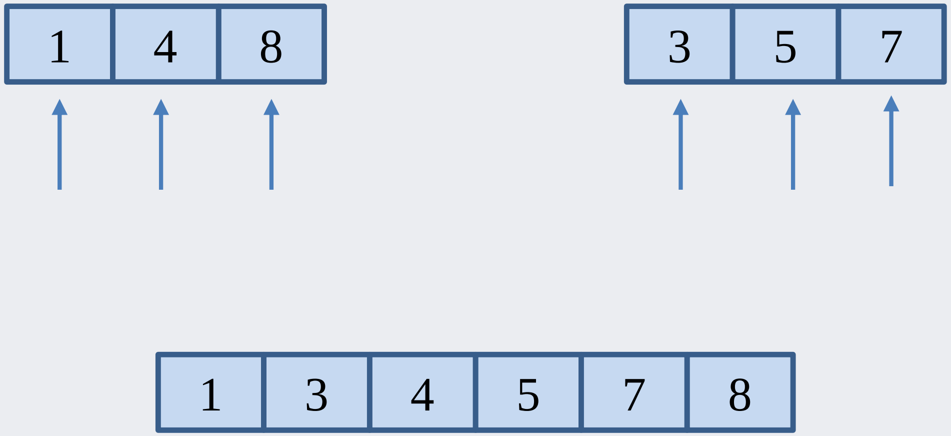
- Merge sort is an example of a divide-and-conquer algorithm that recursively splits the problem into smaller subproblems, solves them, and combines the subproblem solutions to form the overall solution

- Key steps in Merge sort:
  - Split the array into halves
  - Recursively sort each half
  - Merge the two (sorted) halves together to produce a bigger, sorted array
    - the actual sorting occurs in this merge step

  - NOTE: The time to merge two sorted sub-arrays of sizes m and n is linear: $O(m+n)$

Subarray merging

of two sorted subarrays

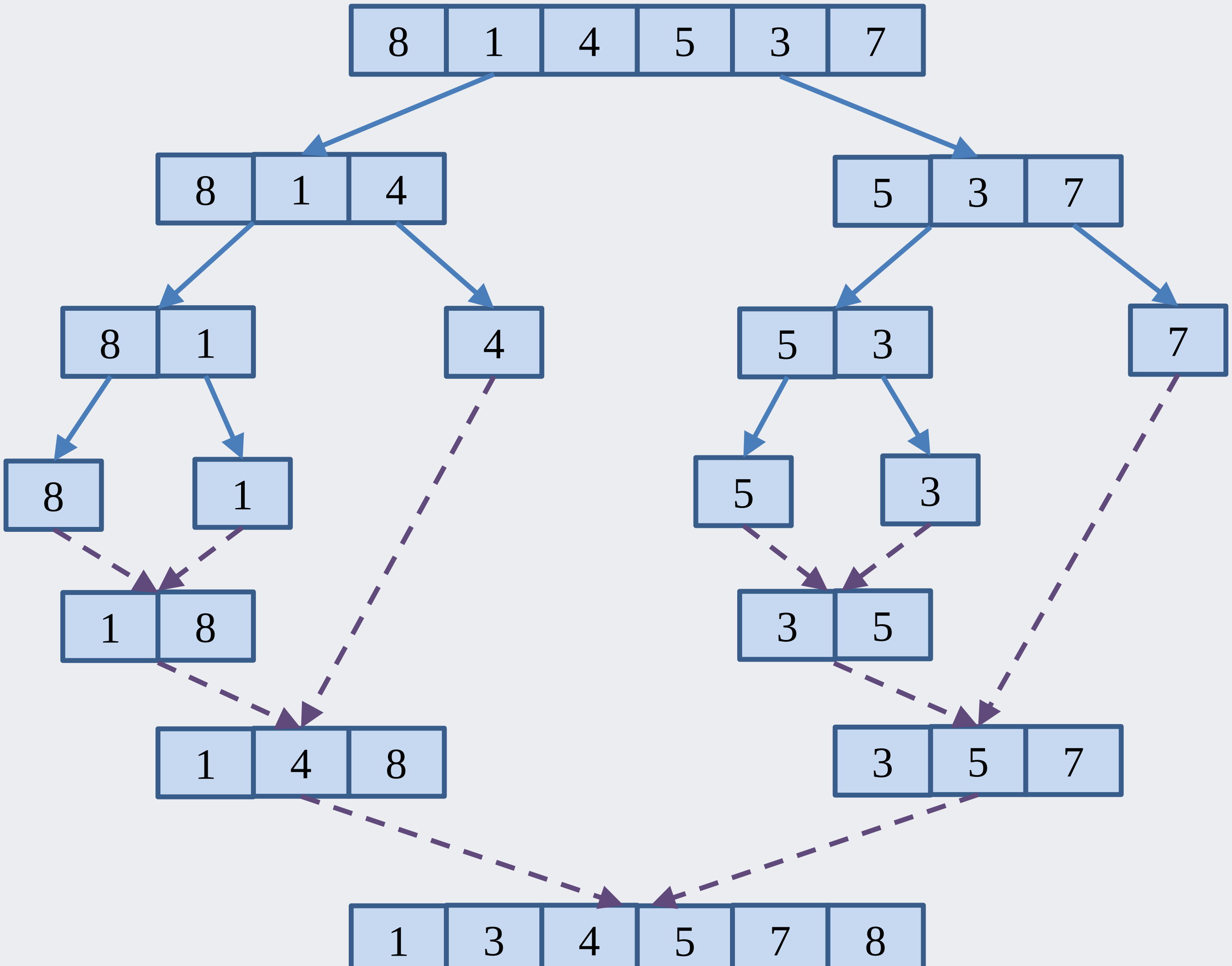


Getting sorted subarrays

- Repeatedly divide arrays in half until each subarray contains a single element
  - an element by itself is already sorted
  - merging two single-element arrays is simply a single comparison

- The merge step copies the subarray halves into a temporary array
  - and the merged elements are copied from the temporary array back to the original array

Merge sort example



Algorithm

```
void merge(int arr[], int low, int mid, int high) {
  int i = low, j = mid+1, index = 0;
  int* temp = (int*) malloc((high – low + 1) * sizeof(int));
  while (i <= mid && j <= high) {
    if (arr[i] <= arr[j])
      temp[index++] = arr[i++];
    else
      temp[index++] = arr[j++];
  }
  if (i > mid) {
    while (j <= high)
      temp[index++] = arr[j++];
  }
  else {
    while (i <= mid)
      temp[index++] = arr[i++];
  }
  for (index = 0; index < high-low; index++)
    arr[low + index] = temp[index];
  free(temp);
}
```

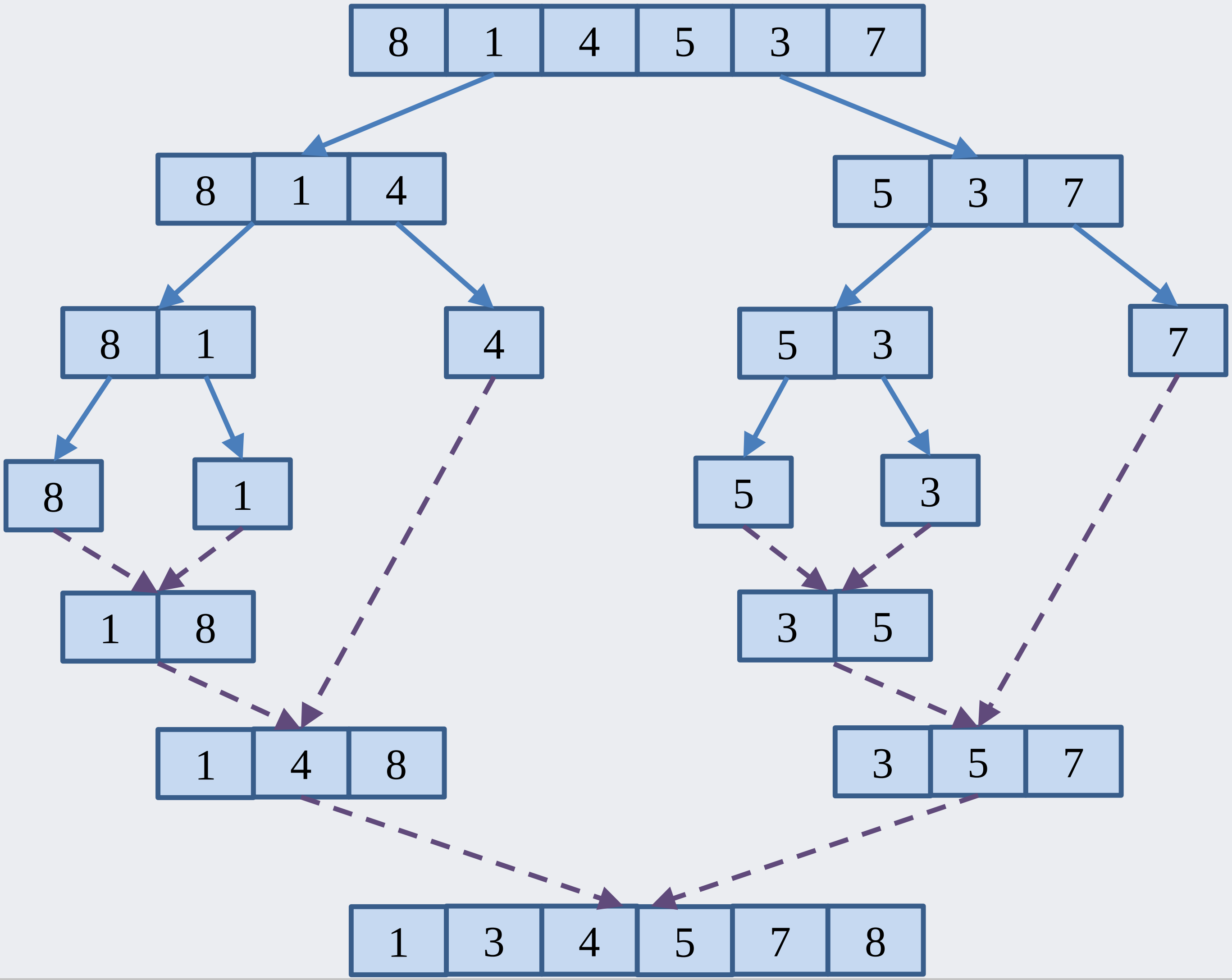
```
void msort(int arr[], int low, int high) {
  int mid;
  if (low < high) {
    // subarray has more than 1 element
    mid = (low + high) / 2;
    msort(arr, low, mid);
    msort(arr, mid+1, high);
    merge(arr, low, mid, high);
  }
}
```

```
void mergeSort(int arr[], int size) {
  msort(arr, 0, size-1);
}
```

1. merging when both subarrays have elements remaining
2. copy remaining elements from non-exhausted subarray
3. copy everything from temporary array into original array

Merge sort example

Again with proper recursion



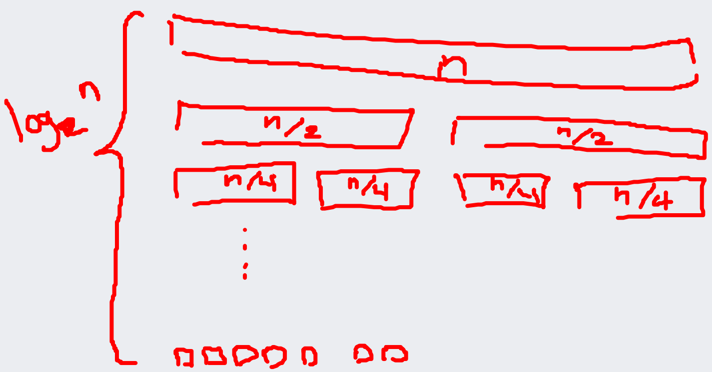
Merge sort analysis

Recursion tree

- How many comparisons are made in the merge step?
  - Worst case: $n - 1$ comparisons
    - need to check every subarray index
  - Best case: $n/2$ comparisons
    - reach the end of one subarray, copy the rest of the second subarray
- Still copying n subarray items in any case
- How many times can the subarray be divided?
  - $\log_2 n$ divisions to reach 1-element subarrays

Overall: $\Theta(n \log n)$, all cases

also one of the best sorting algorithms available



$\Theta(n)$ to merge

$\Theta(n)$ to merge

Merge sort stability

- Stability can be enforced during the merge step at the following places:
  - at the first `while` loop, prioritize duplicates from the left subarray (achieved by the `<=` in the condition)
  - in the two conditional `while` loops (copying remaining subarray elements), copy in the same order encountered
  - in the final `for` loop, keep all the elements in their existing order

Merge sort summary

- External sorting is a term for a class of sorting that can handle massive data sets that do not fit in RAM.
  - Merge sort can be adapted to sort partial data sets brought into RAM from disk
- Merge sort is also highly parallelizable

Name	Best	Average	Worst	Stable	Memory
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	$O(n)$

Quicksort introduction

- Quicksort is an efficient sorting algorithm than either selection or insertion sort
  - It sorts an array by repeatedly *partitioning* it

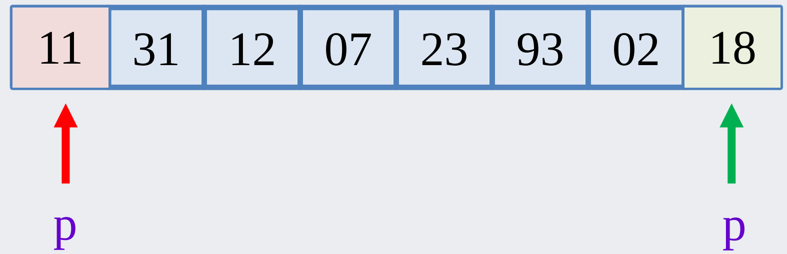
- Partitioning is the process of dividing an array into sections (partitions), based on some criteria
  - Big and small values
  - Negative and positive numbers
  - Names that begin with *a-m*, names that begin with *n-z*
  - Darker and lighter pixels

- Ideally, partitions should be roughly equal in size, but this usually cannot be guaranteed

Array partitioning

Partition array into *small* and *big* values using a partitioning algorithm

Use three indices.
Place two indices, one at each end of the array, call them *low* and *high*.
The third index *p*, start it at low.

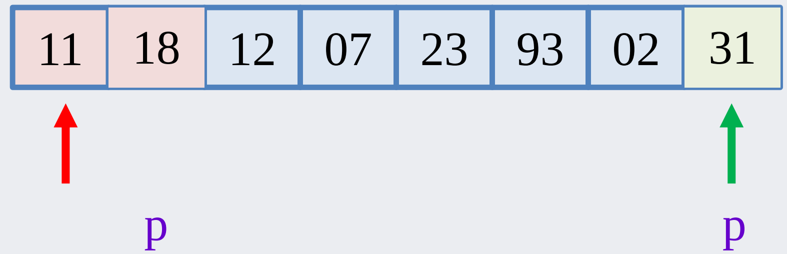


Scan *high* from right to left until arr[*high*] is less than arr[*p*]

arr[*high*] (11) is already less than arr[*p*] (18) so swap them and set *p* to *high*

Array partitioning

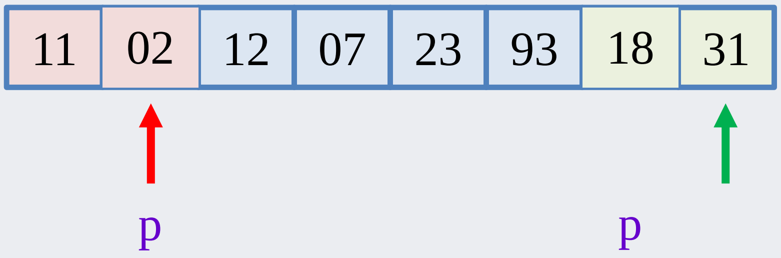
Scan *low* from left to right until arr[*low*] is greater than arr[*p*]



arr[*low*] (31) is greater than arr[*p*] (18) so swap them and set *p* to *low*

Array partitioning

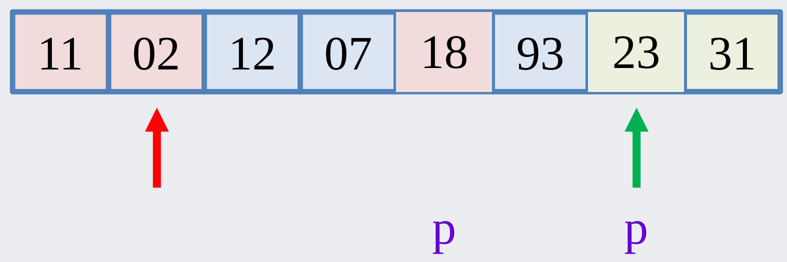
Scan *high* from right to left until arr[*high*] is less than arr[*p*]



arr[*high*] (02) is less than the arr[*p*] (18) so swap them and set *p* to *high*

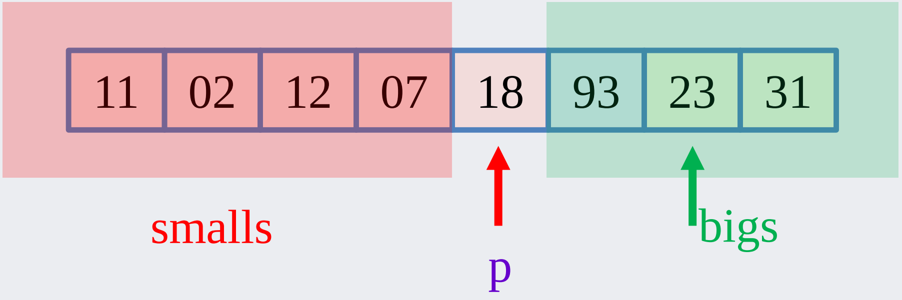
Array partitioning

Scan *low* from left to right until arr[*low*] is greater than arr[*p*]



arr[*low*] (23) is greater than arr[*p*] (18) so swap them and set *p* to *low*

Array partitioning



Scan *high* from right to left until arr[*high*] is less than arr[*p*]
(or *high* equals *p*)

Stop! The index *p* contains the pivot value.
All elements to the left of the pivot have smaller values, all elements to the right of the pivot have larger values (but are not necessarily ordered)

iClicker 11.1

The array below may (or may not) have been partitioned.

5	2	6	27	11	18	25	33	37	62	59	41

How many values could have been a valid pivot used to partition the array?

A. 0
B. 1
C. 2
D. 3
E. 4 or more

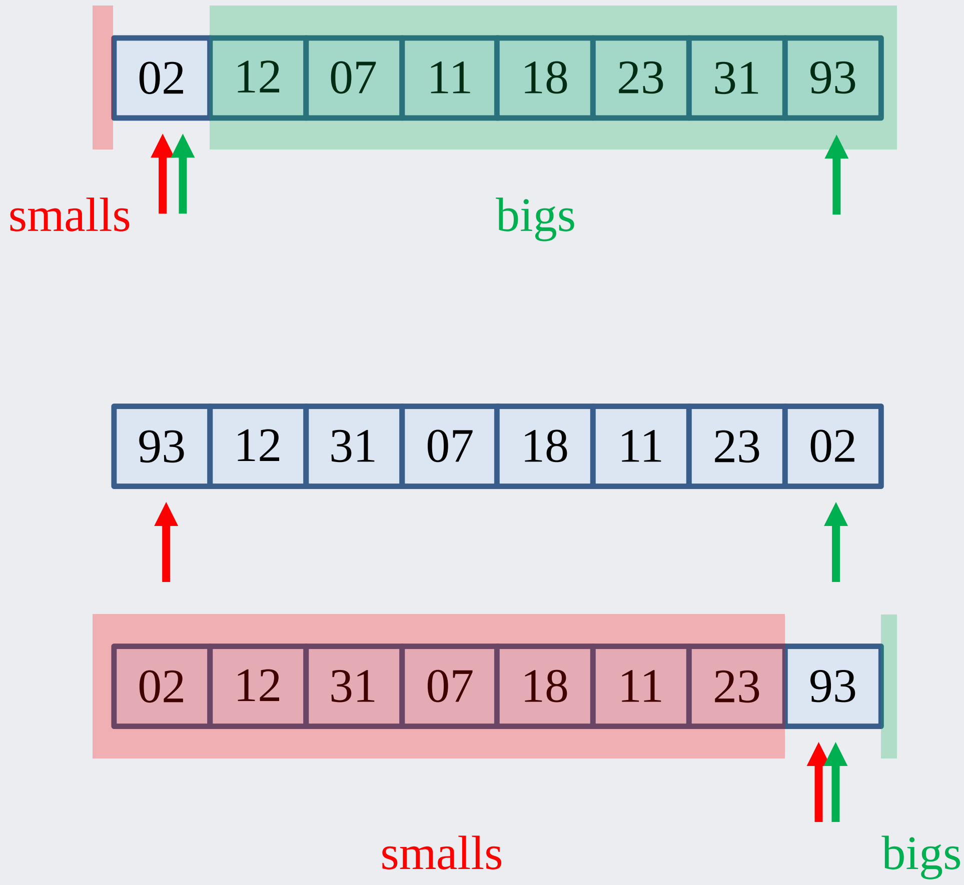
Quicksort overview

- The Quicksort algorithm works by *repeatedly partitioning* an array
- Each time a subarray is partitioned there is
  - A sequence of *small* values,
  - A sequence of *big* values, and
  - A *pivot* value *which is in the correct position*

- Partition the small values, and the big values
  - Repeat the process until each subarray being partitioned consists of just one element
- Ideally, partitions would be halved in size
  - due to unpredictable pivot value, subarray indices also hard to predict

Uneven partitions

- What would the initial partitions look like for these arrays?



Partitions can be unpredictable

Quicksort example

53	61	97	48	11	03	70	47	29	09	36
36	09	29	48	11	03	47	53	70	97	61
03	09	29	11	36	48	47	53	61	70	97
03	09	29	11	36	47	48	53	61	70	97
03	09	29	11	36	47	48	53	61	70	97
03	09	11	29	36	47	48	53	61	70	97
03	09	11	29	36	47	48	53	61	70	97

Quicksort algorithm

```
void qsort(int arr[], int low, int high) {
  int p;
  if (low < high) {
    p = partition(arr, low, high);
    qsort(arr, low, p-1);
    qsort(arr, p+1, end);
  }
}
```

`partition` is where all the comparisons are done, according to the process in the previous slides
See Thareja Ch.14.11 for implementation

Note that there are many different implementations of `partition` in various literature!

```
void quicksort(int arr[], int size) {
  qsort(arr, 0, size-1);
}
```

Quicksort example

Corrected for proper recursion

53	61	97	48	11	03	70	47	29	09	36
36	09	29	48	11	03	47	53	70	97	61
03	09	29	11	36	48	47	53	61	70	97
03	09	29	11	36	47	48	53	61	70	97
03	09	29	11	36	47	48	53	61	70	97
03	09	11	29	36	47	48	53	61	70	97
03	09	11	29	36	47	48	53	61	70	97

Quicksort analysis

- How long does Quicksort take to run?
  - Let's consider the best and the worst case
  - These differ because the partitioning algorithm may not always do a good job
- Let's look at the best case first
  - Each time a sub-array is partitioned the pivot is the exact midpoint of the slice (or as close as it can get)
    - So it is divided in half
  - What is the running time?

Quicksort best case

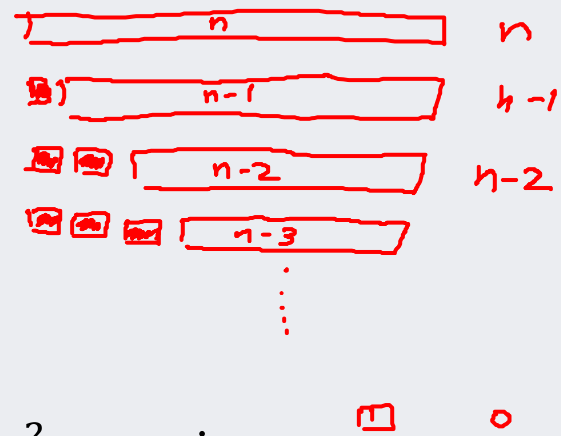
Running time

- Each sub-array is divided in half in each partition
  - Each time a series of sub-arrays are partitioned n (approximately) comparisons are made
  - The process ends once all the sub-arrays left to be partitioned are of size 1
- How many times does n have to be divided in half before the result is 1?
  - $\log_2 n$ times
  - Quicksort performs $n \cdot \log_2 n$ operations in the best case

Same complexity as Merge sort

Quicksort worst case

- Every partition step ends with no values on one side of the pivot
  - The array has to be partitioned n times, not $\log_2 n$ times
  - n comparisons in the first partition step…
  - $n-1$ comparisons in the second step…
  - $n-2$ comparisons in the third step…
  - …
  - $n+(n-1)+(n-2)+\cdots+2+1=\frac{n(n+1)}{2}$
  - So in the worst case Quicksort performs around n^2 operations
- The worst case usually occurs when the array is nearly sorted (in either direction)



As bad as selection sort!

Quicksort average case

- With a large array we would have to be very, very unlucky to get the worst case
  - Unless there was some reason for the array to already be partially sorted
- The average case is much more like the best case than the worst case
- There is an easy way to fix a partially sorted array to that it is ready for Quicksort
  - Randomize the positions of the array elements!

What is the complexity of performing a random scramble of the array?

Merge sort vs Quicksort

and Quicksort summary

- If Quicksort worst case is so bad, why use it?
  - worst case is exceedingly rare and can be easily avoided
  - in practice, faster than Merge sort. Why?
  - can be sorted in-place using $O(\log n)$ stack space
  - also highly parallelizable

Name	**Best**	**Average**	**Worst**	**Stable**	**Memory**
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	$O(n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	challenging	$O(\log n)$

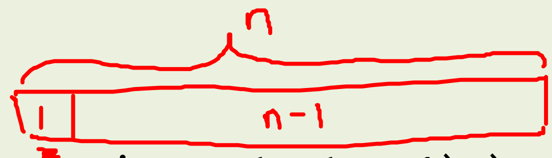
Analysing recursive functions

- Recursive functions are defined in terms of themselves
  - The running time on a given invocation can also be defined in terms of the running times of its own recursive invocations
  - e.g $T(n) = \text{something} + T(\text{a smaller } n) + \ \ldots$

- Like recursion, the running time on some very small input size can be determined immediately (a base case), typically $T(1)$
  - Running time of subproblems can be similarly expressed in terms of running time of its subproblems
  - Repeat the substitutions to establish a pattern
  - Determine the number of substitution levels required to reach a base case
  - Solve for a closed-form expression for running time

Example: Recursive max in an array

```
double arrMax(double arr[], int size, int start) {
  if (start == size - 1)
    return arr[start];
  else
    return max( arr[start], arrMax(arr, size, start + 1) );
}
```



$T(1) \leq b$ amount of work to be done in base case

$T(n) \leq \underbrace{c}_{\text{amount of non-recursive work}} + \underbrace{T(n-1)}_{\text{amount of time needed to solve subproblem of size n-1}}$

- Analysis

  $T(n-1) \leq c + \underbrace{T(n-1-1)}_{T(n-2)}$

  $T(n) \leq c + c + T(n-2)$ (by substitution)

  $T(n-2) \leq c + T(n-3)$

  $T(n) \leq c + c + c + T(n-3)$ (by substitution, again)

  $T(n) \leq k \cdot c + T(n-k)$ getting this to base case size (extrapolating, $0 < k < n$)

  $T(n) \leq (n-1) \cdot c + T(1) = (n-1) \cdot c + b$ for $k = n - 1$

- $T(n) \in O(n)$

Merge sort analysis

Now with even more math!

- Merge sort algorithm
  - Split list in half, sort first half, sort second half, merge together

$T(1) \leq b$

$T(n) \leq 2 \cdot T(n/2) + \underbrace{c \cdot n}_{\text{cost of merging subarrays}}$

$T\left(\frac{n}{2}\right) \leq \boxed{2 \cdot T\left(\frac{n}{2 \cdot 2}\right) + c \cdot \frac{n}{2}}$

- Analysis

$$
\begin{aligned}
T(n) &\leq 2 \cdot T(n/2) + c \cdot n \\
&\leq 2\big(2 \cdot T(n/4) + c(n/2)\big) + cn \\
&= 4 \cdot T(n/4) + c \cdot n + c \cdot n \\
&\leq 4\big(2 \cdot T(n/8) + c(n/4)\big) + c \cdot n + c \cdot n \\
&= 8 \cdot T(n/8) + c \cdot n + c \cdot n + c \cdot n \\
&\leq 2^k \cdot T\left(n/2^k\right) + k \cdot c \cdot n \qquad \text{extrapolating, } 1 < k \leq ? \\
&\leq n \cdot T(1) + c \cdot n \log n \leq n \cdot b + c \cdot n \log n \qquad \text{for } 2^k = n \text{, or } k = \log_2 n
\end{aligned}
$$

$T\left(\frac{n}{4}\right) \leq \boxed{2 \cdot T\left(\frac{n}{8}\right) + c \cdot \frac{n}{4}}$

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log_2 n = \log_2 2^k$

$2^{\log_2 n} \cdot T(1) + \log_2 n \cdot c \cdot n$

- $T(n) \in O(n \log n)$

Binary search

Recurrence analysis

- Inspect midpoint, recursively search left or right half of array
- Base case at a single element

$$T(1) \leq b$$

$$T(n) \leq c + T(n/2)$$

$$T(n/2) \leq c + T(n/4)$$

$$T(n/4) \leq c + T(n/8)$$

$$n/2^k = 1, k = \log_2 n$$

$$\begin{aligned} T(n) &\leq c + c + T(n/4) \\ &\leq c + c + c + T(n/8) \\ &\leq k \cdot c + T\left(n/2^k\right) \\ &\leq c \cdot \log_2 n + T(1) \\ &\leq c \cdot \log_2 n + b \end{aligned}$$

$$T(n) \in O(\log n)$$

Solving exact recurrences

- Same techniques apply but without using inequality
- e.g.
  - equality
  - $T(1) = 2$
  - $T(n) = 2 \cdot T(n-1) + 4$

$T(n-1) = \boxed{2 \cdot T(n-2) + 4}$

$T(n) = 2 \cdot (2 \cdot T(n-2) + 4) + 4$

$T(n-2) = 2 \cdot T(n-3) + 4$

$= 2 \cdot 2 \cdot T(n-2) + 2 \cdot 4 + 1 \cdot 4$

$= 2 \cdot 2 \cdot (2 \cdot T(n-3) + 4) + 2 \cdot 4 + 1 \cdot 4$

$= 2 \cdot 2 \cdot 2 \cdot T(n-3) + 2^2 \cdot 4 + 2^1 \cdot 4 + 2^0 \cdot 4$

$= 2^k \cdot T(n-k) + 4 \cdot \sum_{i=0}^{k-1} 2^i$

let $k = n-1$

$= 2^{n-1} \cdot T(1) + 4 \cdot \sum_{i=0}^{n-2} 2^i$

1 1 1 1

1 0 0 0 0 0

2^n

$= \underbrace{2^{n-1} \cdot 2}_{2^n} + \underset{2^2}{4} \cdot (2^{n-1} - 1)$

$2^{n+1} - 4$

Getting recurrences

for algorithms

base case size
(usually some specified constant)

base case cost
(usually some unspecified constant, but could be something else)

$$\begin{cases} T(\underline{\hspace{3em}}) \le \underline{\hspace{3em}} \\ T(n) \le \underline{\hspace{3em}} \cdot T(\underline{\hspace{3em}}) + \underline{\hspace{4em}} \end{cases}$$

How many subproblems
(if subproblems are all the same size)

Size of each subproblem
(a function of n, and must be $< n$)

Cost of non-recursive operations
(constant, or some function of n)

Readings for this lesson

- Thareja
  - Chapter 14.9, 14.8, 14.10, 14.11 (Selection sort, Insertion sort, Merge sort, Quicksort)