some slides are adapted from the slides of David Luebke

Sorting

Sorting. Given n elements, rearrange in ascending order.

Obvious sorting applications.
- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

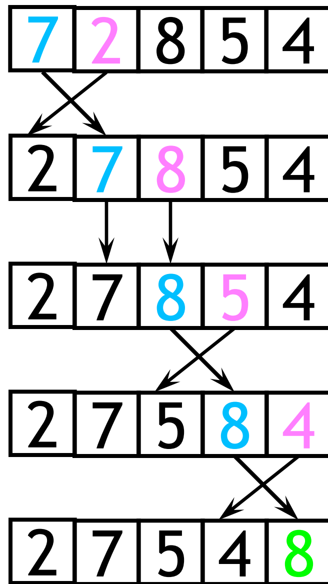
Problems become easier once sorted.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

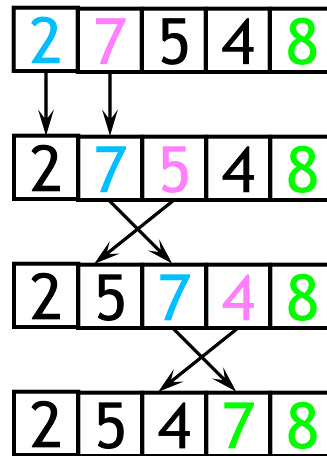
Non-obvious sorting applications.
- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- . . .

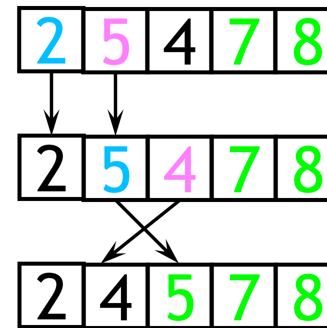
Bubble sort

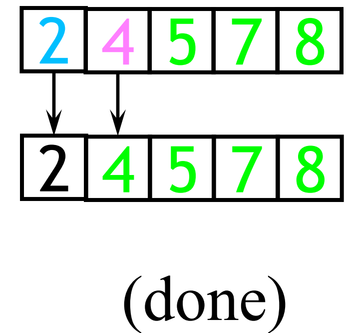
- Compare each element (except the last **one**) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last ***two***) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Continue as above until no elements are swapped during a scan

Example of bubble sort









Analysis of bubble sort

- n operations in the first scan
- (n-1) operations in the second scan
- .
- .
- .
- (n-(i-1)) operations in i-th scan

- Adding the number of operations over all scans we have
  - $n+(n-1)+(n-2) + \ldots + 1 = O(n^2)$

Analysis of bubble sort

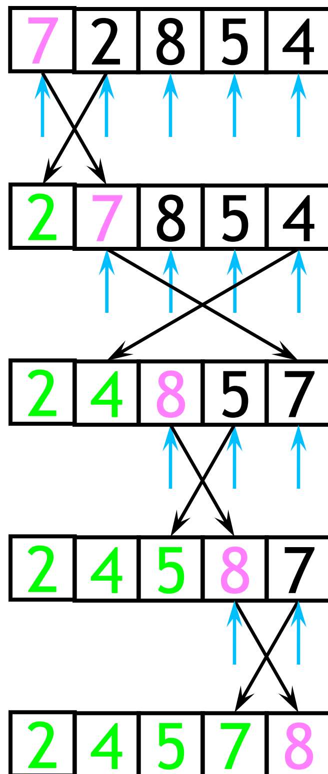
- In the best case (sorted list) we just scan the list once: $O(n)$ time
- In a list sorted from the largest to the smallest element we spend $\Omega(n^2)$
- Algorithm has $\theta(n^2)$ worst case

Selection sort

Given an array of length **n**,

- Search elements **1** through **n** and select the smallest
  - Swap it with the element in location 1
- Search elements **2** through **n** and select the smallest
  - Swap it with the element in location **2**
- Search elements **3** through **n** and select the smallest
  - Swap it with the element in location **3**
- Continue in this fashion until there's nothing left to search

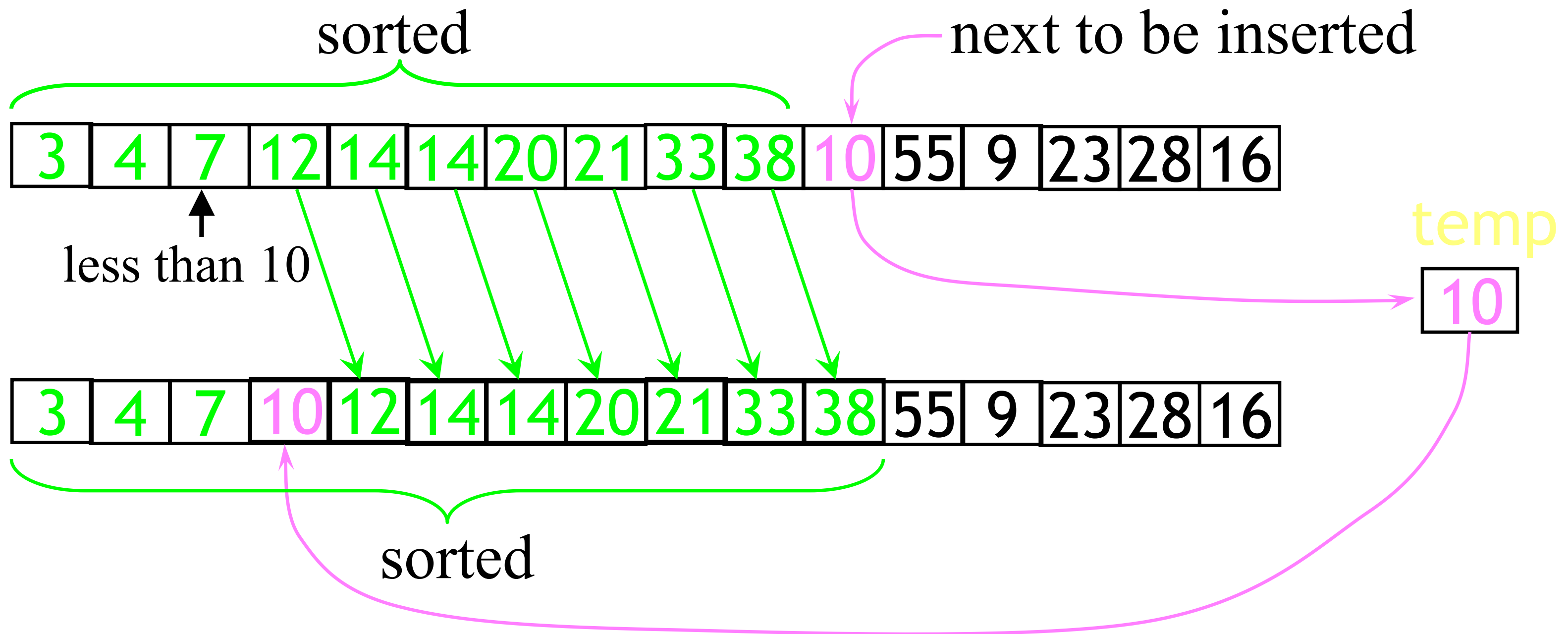
Example and analysis of selection sort



Analysis:

- The outer loop executes **n-1** times
- The i-th internal loop executes (n-i) operations
- Work done in the inner loop is constant (swap two array elements)
- $\theta(n^2)$ in the worst case (always)

One step of insertion sort



Implemented as a vector

- We run once through the outer loop, inserting each of n elements; this is a factor of **n**
- In the i-th loop we move at most **n-i** elements ➔ insertion sort is **O(n²)**
- For the completely unsorted sequence it executes **n²** operations ➔ insertion sort is Ω(**n²**)

Implemented as an inserted list

- Finding the position to insert requires O(n) time
- Insertion can be done in O(1) time

Summary

Bubble sort, selection sort, and insertion sort are all **O(n^2)**

As we will see later, we can do much better than this with somewhat more complicated sorting algorithms

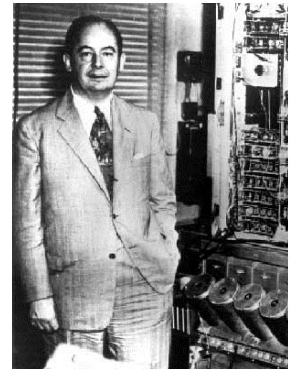
Within **O(n^2)**,

- Bubble sort is very slow, and should probably never be used for anything
- Selection sort is intermediate in speed
- Insertion sort is usually the fastest of the three--in fact, for small arrays (say, 10 or 15 elements), insertion sort is faster than more complicated sorting algorithms

Selection sort and insertion sort are "good enough" for small arrays

5.1 Mergesort

Mergesort



Jon von Neumann (1945)

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R		I	T	H	M	S	divide	$O(1)$
A	G	L	O	R		H	I	M	S	T	sort	$2T(n/2)$

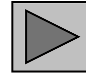
A	G	H	I	L	M	O	R	S	T	merge	$O(n)$

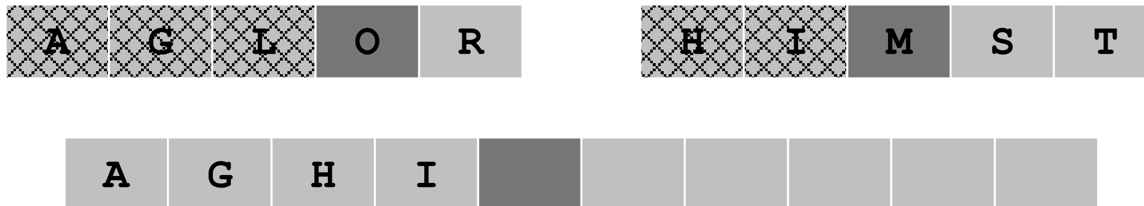
Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.





Challenge for the bored. In-place merge. [Kronrud, 1969]

↑ using only a constant amount of extra storage

A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

Mergesort recurrence.

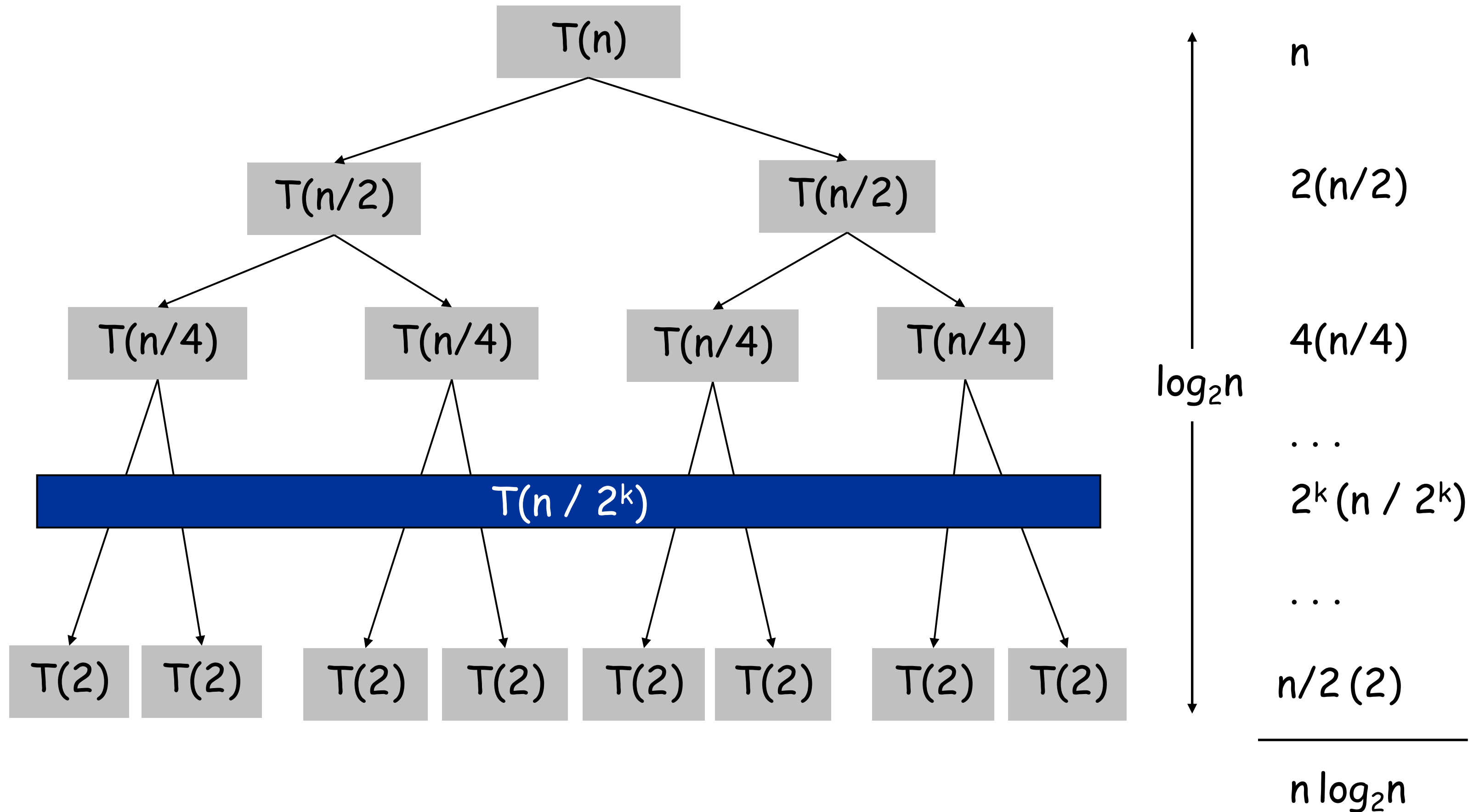
$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(n \log_2 n)$.

Proofs. We describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace ≤ with =.

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Proof by Telescoping

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

↑ assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. For n > 1:

$$\begin{aligned} \frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\ &= \frac{T(n/2)}{n/2} + 1 \\ &= \frac{T(n/4)}{n/4} + 1 + 1 \\ &\cdots \\ &= \frac{T(n/n)}{n/n} + \underbrace{1 + \cdots + 1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

Proof by Induction

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

↑ assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: $n = 1$.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n\left(\log_2(2n) - 1\right) + 2n \\ &= 2n \log_2(2n) \end{aligned}$$

Analysis of Mergesort Recurrence

Claim. If T(n) satisfies the following recurrence, then $T(n) \le n \lceil \lg n \rceil$.

$\uparrow$ $\log_2 n$

$$
T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}
$$

Pf. (by induction on n)

- Base case: n = 1.
- Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.
- Induction step: assume true for 1, 2, ... , n–1.

$$
\begin{aligned}
T(n) &\le T(n_1) + T(n_2) + n \\
&\le n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \\
&\le n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n \\
&= n \lceil \lg n_2 \rceil + n \\
&\le n (\lceil \lg n \rceil - 1) + n \\
&= n \lceil \lg n \rceil
\end{aligned}
$$

$$
\begin{aligned}
n_2 &= \lceil n/2 \rceil \\
&\le \left\lceil 2^{\lceil \lg n \rceil} / 2 \right\rceil \\
&= 2^{\lceil \lg n \rceil} / 2 \\
\Rightarrow \lg n_2 &\le \lceil \lg n \rceil - 1
\end{aligned}
$$

Quicksort

- Sorts in place
- Sorts $O(n \lg n)$ in the average case
- Sorts $\theta(n^2)$ in the worst case

So why would people use it instead of mergesort/heapsort?

Worst case is quite rare and constants are low!

Quicksort

A divide-and-conquer algorithm

- The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
  - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
- The subarrays are recursively sorted by calls to quicksort

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

- q is the index of the end of the left partition

Partition

Clearly, all the action takes place in the `partition()` function

- Rearranges the subarray in place
  - Two subarrays
  - All values in first subarray $\leq$ all values in second
- Returns the index where the left partition ends

How do you suppose we implement this function?

Partition

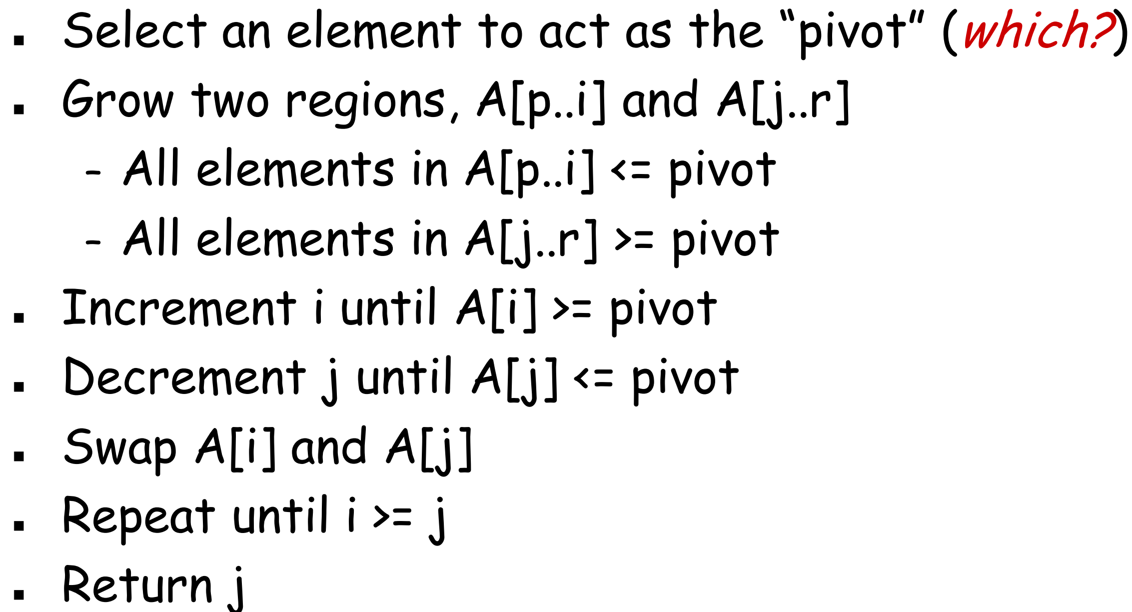
We can use an auxiliary vector.

- Select a pivot
- In the left side we put all elements smaller than or equal to the pivot
- In the right side we put all elements larger than the pivor

Linear time but not in place

In Place Partition (In Words)

Partition(A, p, r):



In Place Partition (Code)

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

Illustrate on
A = {5, 3, 2, 6, 4, 1, 3, 7};

What is the running time of ***partition()****?*

Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

`partition()` *runs in O(n) time*

Analyzing Quicksort: Worst case

T(n): QuickSort worst case complexity for instance of size n.

$T(1) = c$
$T(n) \le \max_{i=1...n} \{ T(n - i) + T(i-1) + cn\}$

Provar por indução que $T(n) \le cn^2$

- Base é válida, n=1.

- Seja i* o valor de i que maximiza a expressão. Segue da indução que $T(n) \le c(n-i^*)^2+c(i^*-1)^2+cn$.

- O lado direito é uma parábola em i*. Portanto, ela atinge o máximo em i*=1 ou i*=n. Testando os dois valores estabelecemos a hipótese de indução

Analyzing Quicksort: Worst case

T(n): QuickSort worst case complexity for instance of size n.

- The height of execution tree of QuickSort is at most n.
- At each level of the tree the algorithm spends O(n) due to the partition precedure
- The algorithm spends at most cn^2 operations

QuickSort is $O(n^2)$

Analyzing Quicksort: worst case

- Para uma entrada ordenada as partições sempre são desbalanceadas

  - O algoritmo gasta $1+2+...+n = n^2$

QuickSort is $\Omega(n^2)$

Analyzing Quicksort: Best Case

- B(n): complexidade melhor caso para o QuickSort
- Se o pivot sempre divide a lista corrente em duas listas de mesmo tamanho temos $B(n)= n+ 2B(n/2)$
  - Desenvolvendo temos B(n) é O(n log n)

Analyzing Quicksort: Average Case

- A(n): complexidade de caso médio para o QuickSort assumindo todas as permutações equiprováveis
- So partition generates splits (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0) each with probability 1/n

$$A(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[A(k) + A(n-1-k)\right] + \Theta(n)$$

- Desenvolvendo temos A(n) é O(n log n)

Improving Quicksort

The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input

Book discusses two solutions:

- Randomize the input array, OR
- *Pick a random pivot element*

How will these solve the problem?

- By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Exercício

Descreva um algoritmo com complexidade O(n log n) com a seguinte especificação

Entrada: Uma lista de n números reais

Saida: SIM se existem números repetidos na lista e NÃO caso contrário

Exercício - Solução

Ordene a lista lista L
Para i=1 até |L|-1
Se L[i]=L[i+1 **Devolva** SIM
Fim Para
Devolva NÃO

- ANÁLISE
  - A ordenação da lista L requer O (n log n) utilizando o MergeSort
  - O loop **Para** requer tempo linear

Exercício

Descreva um algoritmo com complexidade $O(n \log n)$ com a seguinte especificação

Entrada: conjunto *S* de *n* números reais e um número real *x*

Saida: SIM se existem dois elementos em *S* com soma x e NÃO caso contrário

Exercício - Solução

L <- conjunto S em ordem crescente
Enquanto a lista L tiver mais que um elemento **faça**
  Some o menor e o maior elemento de L
  Se a soma é x
    Devolva SIM
  Se a soma é maior que x
    Retire o maior elemento de L
  Se a soma é menor que x
    Retire o menor elemento de L
Fim Enquanto
Devolva NÃO

Exercício - Solução

L <- conjunto S em ordem crescente

Enquanto a lista L tiver mais que um elemento **faça**

Some o menor e o maior elemento de L

Se a soma é x

Devolva SIM

Se a soma é maior que x

Retire o maior elemento de L

Se a soma é menor que x

Retire o menor elemento de L

Fim Enquanto

Devolva NÃO

- ANÁLISE
  - A ordenação do conjunto S requer O (n log n) utilizando o o MergeSort
  - O loop enquanto requer tempo linear

Sorting So Far

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - $O(n^2)$ worst case

Sorting So Far

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key < children's keys
  - O(n lg n) worst case
  - Sorts in place

Sorting So Far

- MergeSort
  - θ(n lg n) worst case
  - It does not sort in place

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
  - Fast in practice
  - $\theta(n^2)$ worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort, $O(n \log n)$ expected time

How Fast Can We Sort?

- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: The running time of any comparison sort algorithm is $\Omega(n \lg n)$
    - A comparison sort must do at least n/2 comparisons (*why?*)
    - What about the gap between n/2 and $\Omega(n \lg n)$?

Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
  - A decision tree represents the comparisons made by a comparison sort. Every thing else ignored
  - (Draw examples on board, Bubble sort with 3 elements)
- *What do the leaves represent?*
- *How many leaves must there be?*

Decision Trees

- Decision trees can model comparison sorts. For a given algorithm:
  - One tree for each *n*
  - Tree paths are all possible execution traces
  - *What's the longest path in a decision tree for insertion sort? MergeSort?*
- *What can we say about the height of a decision tree for sorting n elements?*

Lower Bound For Comparison Sorting

- Theorem: The height of any decision tree that sorts *n* elements is $\Omega(n \lg n)$
  - Any decision tree has at least n! leaves
  - The height of a decision tree with n! leaves is at least log(n!)
  - log(n!)>=n/2 log n -2n (analysis slide)

Lower Bound For Comparison Sorts

- Thus the time to comparison sort n elements is $\Omega(n \lg n)$
- Corollary: MergeSort is asymptotically optimal comparison sort

Sorting In Linear Time

- Counting sort
  - No comparisons between elements!
  - ***But***...depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1.. k*
  - The algorithm:
    - Input: A[1..n], where A[j] $\in$ {1, 2, 3, …, k}
    - Output: B[1..n], sorted (notice: not sorting in place)
    - Also: Array C[1..k] for auxiliary storage

Counting Sort

```
1       CountingSort(A, B, k)
2               for i=1 to k
3                       C[i]= 0;
4               for j=1 to n
5                       C[A[j]] += 1;
6               for i=2 to k
7                       C[i] = C[i] + C[i-1];
8               for j=n downto 1
9                       B[C[A[j]]] = A[j];
10                      C[A[j]] = C[A[j]]- 1;
```

Work through example: A={4 1 3 4 3}, k = 4

Counting Sort

```
1  CountingSort(A, B, k)
2       for i=1 to k
3            C[i]= 0;
4       for j=1 to n
5            C[A[j]] += 1;
6       for i=2 to k
7            C[i] = C[i]+C[i-1];
8       for j=n downto 1
9            B[C[A[j]]] = A[j];
10           C[A[j]] = C[A[j]]- 1;
```

Initialize

C[i] stores the number of elements equals to i,i =1,…,k

C[i] stores the number of elements smaller or equal to i,i =1,…,k

The position of element A[j] in B is equal to the number of integers that are smaller than or equal to A[j] which is C[A[j]]

Counting Sort

```
1     CountingSort(A, B, k)
2           for i=1 to k
3                 C[i]= 0;
4           for j=1 to n
5                 C[A[j]] += 1;
6           for i=2 to k
7                 C[i] = C[i] + C[i-1];
8           for j=n downto 1
9                 B[C[A[j]]] = A[j];
10                C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

Counting Sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$!
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is *stable*
    - If x and y are two identical numbers and x is before y in the input vector then x is also before y in the output vector (Important for Radix sort)

Counting Sort

- Cool! *Why don't we always use counting sort?*
- Because it depends on range k of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, k too large ($2^{32} = 4{,}294{,}967{,}296$)

Radix Sort

- Intuitively, you might sort on the most significant digit, then the second m.s.d., etc.
- Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

  - Example: Fig 9.3

Radix Sort

- (329,457,657,839,436,720)

Step 1: third digit

(720, 436, 457,657, 329,839)

Step 2: second digit

(720, 329,436, 839,457,657)

Step 3: first digit

(329,436,457,657, 720, 839)

Radix Sort

- *Can we prove it will work?*
- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits {j: j<i}are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and $k=O(n)$, takes $O(n)$ time

Radix Sort

Lemma 8.4 (Cormen). Given n b-bits numbers and any positive integer r<=b, RadixSort correctly sort these numbers in O($(b/r)(n+2^r)$) time.

Proof Sketch. We can split a b-bits number in b/r groups of r bits. Each group of r-bits is viewed as a digit in the range $[0, 2^r-1]$.

Radix Sort

Lemma 8.4 (Cormen). Given n b-bits numbers and any positive integer r<=b, RadixSort correctly sort these numbers in O($(b/r)(n+2^r)$) time.

It is desirable to select r so that $(b/r)(n+2^r)$ is minimized

- If b< log n then r=b is an optimal choice ➔
  O(n) time
- If b>= log n then r=log n is an optimal choice ➔
  O(bn /logn) time

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical $O(n \lg n)$ comparison sort
  - Requires approx $\lg n = 20$ operations per number being sorted

Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice