Lab 6
CIS2520, F11

Mohammad Naeem
School of Computer Science (SOCS), UoG

Topics

- time complexity of
  - bubble sort
  - merge sort
  - binary search

Time complexity of algorithms

- running an implementation on a computer
- apply algorithmic analysis
  - count primitive operations
  - for a given size of input
- growth rate of an algo
  - rate at which running time grows as input grows

Time complexity of algorithms

- **constant** growth rate

```
int findFirstElement(int[] a) {

   int firstElement = a[0];
   return firstElement;
}
```

n = input size
f(n) = worst-case running time

$\boxed{f(n)=k}$

where k is a constant

growth rate is order 1 or $O(1)$

Time complexity of algorithms

- **linear** growth rate

```
int findSmallElement(int[] a){

  int smElement = a[0];

  for(int i=0; i<n ; i++)
    if(a[i] < smElement)
      smElement=a[i];
  return smElement;

}
```

n = input size
f(n) = worst-case running time

$$\boxed{f(n)=k_1 n+k_0}$$

where k_1 and k_0 are constants

growth rate is order **n or $O(n)$**

Time complexity of algorithms

- **quadratic** growth rate

```
int findSmallElement(int[][] a){
  int smElement = a[0][0];

  for (int i=0; i<n ; i++)
   for(int j=0; j<n ; j++)
    if(a[i][j] < smElement)
      smElement=a[i][j];

  return smElement;
}
```

n = input size
f(n) = worst-case running time

$$\boxed{f(n)=k_2 n^2+k_1 n+k_0}$$

where k_2, k_1, k_0 are constants

growth rate is order **n^2 or $O(n^2)$**

Time complexity of algorithms

$\boxed{\mathbf{f(n)} = n^3}$ cubic

$\boxed{\mathbf{f(n)} = \log n}$ logarithmic

$\boxed{\mathbf{f(n)} = n \log n}$ linearithmic

which is the best?

Time complexity of algorithms

extern void **Initialize** (Stack *S);
extern void **Push** (Item I, Stack *S);
extern void **Pop** (Stack *S);
extern int **Full** (Stack *S);
extern int **Empty** (Stack *S);
extern int **Size** (Stack *S);
extern void **Top** (Stack *S, Item *I);
extern void **Destroy** (Stack *S);

which is constant, linear, etc.?

Time complexity of algorithms

- **constant time** $O(1)$
  - fixed number of steps
  - push, pop (stacks)
  - enqueue, dequeue (queues)
- **linear** $O(n)$
  - proportional to the problem size
  - sequential search in an unsorted list
  - displaying all elements in an unsorted list
- **quadratic** $O(n^2)$
  - prop to problem size (but expensive relatively)
  - bubble sort
  - finding duplicates in an array

Time complexity of algorithms

- **logarithmic** $O(\log n)$
  - grows logarithmically
  - binary search
  - insert and remove in binary tree
- **linearithmic** $O(n \log n)$
  - more expensive than logrithmic but still better
  - quick sort, merge sort
- **exponential** $O(a^n)$ $(a > 1)$
  - very expensive, un-manageable computationally
  - recursive fibonnaci
  - finding all permutations of n numbers

Time complexity of algorithms

- In a neighborhood of infinity:

  1 < log n < n < n log n < n^2 < 2^n < n! < n^n
- in $\log n$, the base doesn't matter
- given growth rate,

  f(n) $= a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n^1 + a_0$

  growth determined by the fastest growing term

  f(n) is $O(n^k)$

Time complexity of algorithms

{perform any statement S_1}	**O(1)**
for (i=0; i < **n**; i++)	
{	
{perform any statement(s) S_2}	**O(n)**
{run through another loop n times}	**O(n^2)**
}	

Total Execution Time: $1 + n + n^2$ therefore, **O(n^2)**

Time complexity of algorithms

- **running time *f(n)*:** effect of algo design
- **example**

given :

int[] a = {3, 4, 1, 3, 2, 7, 4, 4, 2, 6, 1, 4}

required :

sums for contiguous subsequence of length 5

array size n=12
subsequence length m=5

no of subsequence = n – m + 1 = 12-5 +1 = 8

 # Using Brute force algorithm:
S0 = a[0] + a[1] + a[2] + a[3] + a[4] = 3+4+1+3+2=13
S1 = a[1] + a[2] + a[3] + a[4] + a[5]=4+1+3+2+7=17
S2 = a[2] + a[3] + a[4] + a[5] + a[6]=1+3+2+7+4=17
S3 = a[3] + a[4] + a[5] + a[6] + a[7]=3+2+7+4+4=20
S4 = a[4] + a[5] + a[6] + a[7] + a[8]=2+7+4+4+2=19
S5 = a[5] + a[6] + a[7] + a[8] + a[9]=7+4+4+2+6=23
S6 = a[6] + a[7] + a[8] + a[9] + a[10]=4+4+2+6+1=17
S7 = a[7] + a[8] + a[9] + a[10] + a[11]=4+2+6+1+4=17

Using Brute force algorithm total number of additions =8*4=32.

 # Using previous subsequence(S k+1=Sk + a[k+m] – a[k])
S0 = a[0] + a[1] + a[2] + a[3] + a[4] = 3+4+1+3+2=13
S1 = S0 + a[5] -a[0] = 13+7-3=17
S2 = S1 + a[6] - a[1] = 17+4-4=17
S3 = S2+ a[7] - a[2] = 17+4-1=20
S4 = S3+ a[8] – a[3] = 20+2-3=19
S5 = S4+ a[9] – a[4] = 19+6-2= 23
S6 = S5 + a[10] – a[5] = 23+1-7= 17

S7 = S6+ a[11] – a[6] = 17+4-4= 17
Total number of additions =18

bubble sort: example

First Pass:

(**5 1** 4 2 8) → (**1 5** 4 2 8) compares the first two elements, and swaps them.
(1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) → (1 4 2 **5 8**), already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) → (**1 4** 2 5 8)
(1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Third Pass:

(**1 2** 4 5 8) → (**1 2** 4 5 8)
(1 **2 4** 5 8) → (1 **2 4** 5 8)
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

bubble sort: algorithm

```
function bubbleSort (A)

  n = A.length
  repeat
      swapped = false
      for i = 1 to n-1
          if A[i-1] > A[i]
              swap A[i-1] and A[i]
              swapped = true
      n = n - 1
  until not swapped
```

bubble sort: complexity

```
function bubbleSort (A)
  n = A.length
  repeat
     swapped = false
     for i = 1 to n-1
          if A[i-1] > A[i]
             swap A[i-1] and A[i]
             swapped = true
      n = n - 1
   until not swapped
```

Σ O(n)   O(n)   O(1)

$$\sum \mathbf{O(n) = O(1 + 2 + 3 + \cdots + (n-1))}$$

$$= \boldsymbol{O(\frac{n(n-1)}{2})}$$

$$= \mathbf{O(\,n^2)}$$

merge sort , binary search

divide-and-conquer algorithms

- **generic pattern** – take a problem of size n by
  - diving into sub-problems of size n/b
  - recursively solve the sub-problems
  - combine the sub-solutions in $O(n^d)$ for some $d > 0$ into final solution

- **running time captured by,**

$$T(n) = \underbrace{a\,T(\lceil n/b \rceil)}_{} + O(n^d)$$

time for the whole problem

time for the sub-problem

time for combining sub-solutions into result solution

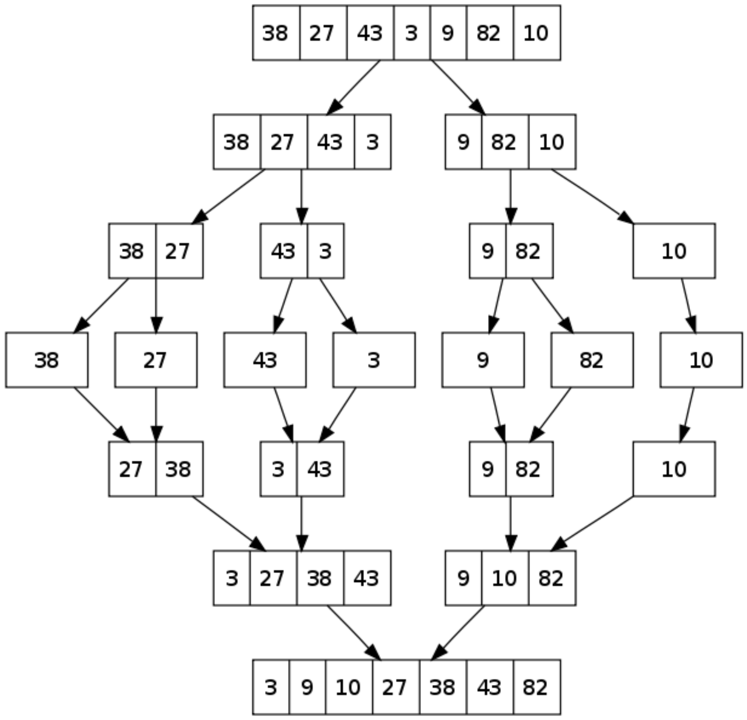
merge sort: algorithm

```
function merge_sort (m)                  T(n)

   if m.length ≤ 1 return m              O(1)
   middle = m.length / 2                 O(1)

   let left be the sublist of m from m[0] to m[middle-1]

   let right be the sublist of m from m[middle] to m[m.length-1]

   left = merge_sort(left)               T(n/2)
   right = merge_sort(right)             T(n/2)
   result = merge(left, right)           O(n)

return result
```

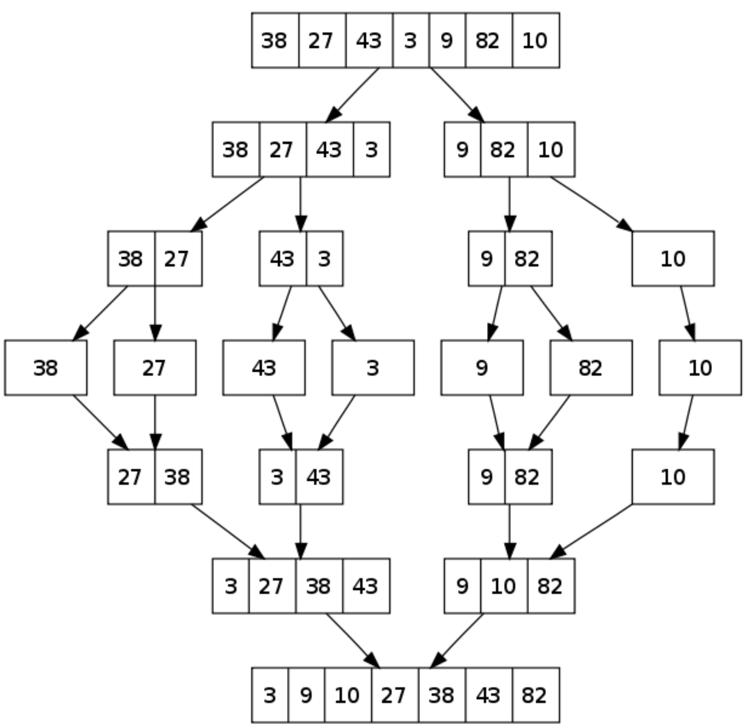

merge sort: algorithm

```
function merge (left, right)
   let result be an empty list
   while left.length > 0 or right.length > 0
      if left.length > 0 and right.length> 0
         if left[0] ≤ right[0]
            append left[0] to result
            remove left[0] from left
         else
            append right[0] to result
            remove right[0] from right
      elseif left.length > 0
         append left to result and empty left
      else
         append right to result and empty right
   return result
```

merge sort: example



merge sort: example



divide

merge

recursively solve the
sub-problems

merge sort: running time analysis

- ***n*** number of items to be merge-sorted, $n = 2^k$
- time required to break the list into two half-lists (**ignore**)
- time $\boldsymbol{T(n/2)}$ -to merge-sort the left sub-list (L)
- time $\boldsymbol{T(n/2)}$ -to merge-sort the right sub-list (R)
- time - $\boldsymbol{bn}$ to merge the L and R into the final list

(*b* cost single merge operation)

$$T(n) = \begin{cases} O(1) & \text{if n} \leq 1 \\ 2T(n/2) + bn & \text{if n} > 1 \end{cases}$$

base case

recurrence relation---- an equation describing a function in terms of its values on smaller inputs

merge sort: running time analysis

assume $n = 2^k$ for some k, where n no of values in the list

$T(n) = 2T(n/2) + bn$ general case for *n > 1*

expand $T(n/2)$ i.e., put $T(n/2) = 2\,T((n/2)/2) + bn\;/\;2$

$$= 2[2\,T((n/2)/2) + b(n/2)] + bn)$$

$$= 2^2T(n/2^2) + 2\,bn/2\; + \;bn$$

$$= 2^2T(n/2^2) + 2^1bn/2^1 + 2^0bn\;/\;2^0$$

merge sort: running time analysis

$T(n) = 2^2 T(n/2^2) + 2^1 bn/2^1 + 2^0 bn \,/\, 2^0$

expand $T(n/2^2\ )$

$= 2^2[2T((n/2^2)/2) + b(\mathrm{n}/2^2)] + 2^1 bn/2^1 + 2^0 bn \,/\, 2^0$

$= 2^3 T(n/2^3) + 2^2 b\, n \,/\, 2^2 + 2^1 bn \,/\, 2^1 + 2^0 b\, n/2^0$

by induction

$= 2^i T\left(n/2^i\right) + 2^{i-1} b\, n \,/\, 2^{i-1} + \cdots + 2^1 bn \,/\, 2^1 + 2^0 b\, n/2^0$

$n = 2^k, i$ eventually becomes

$T(n) = 2^k T\left(n/2^k\right) + 2^{k-1} bn \,/\, 2^{k-1} + \cdots + 2^1 bn \,/\, 2^1 + 2^0 bn \,/\, 2^0$

…

$T(n) = 2^i T\left(n/2^i\right) + 2^{i-1} b\, n \,/\, 2^{i-1} + \cdots + 2^1 bn \,/\, 2^1 + 2^0 b\, n/2^0$

now $n = 2^k$ for some value of k, eventually $i = k$

$= \underbrace{2^k T\left(n/2^k\right)}_{} + 2^{k-1} bn \,/\, 2^{k-1} + \cdots + 2^1 bn \,/\, 2^1 + 2^0 bn \,/\, 2^0$

put $2^k = \ n$

$= n\, T(1) + 2^{k-1} bn \,/\, 2^{k-1} + \cdots + 2^1 bn \,/\, 2^1 + 2^0 bn \,/\, 2^0$

as $T(1) = 1$ base case

$= n \ + \underbrace{2^{k-1} bn \,/\, 2^{k-1} + \cdots + 2^1 bn \,/\, 2^1 + 2^0 bn \,/\, 2^0}_{}$

$$= n + \underbrace{2^{k-1}bn \,/\, 2^{k-1} + \cdots + 2^{1}bn \,/\, 2^{1} + 2^{0}bn \,/\, 2^{0}}$$

, as $n = 2^k$

$$= n + bn \sum_{j}^{k-1} (2^j / 2^j)$$

$$= n + bn \sum_{j}^{k-1} (2^j / 2^j)$$

as $\sum_{j=0}^{k-1} (2^j/2^j) = \sum_{j=0}^{k-1} 1 = k$ so

$$= n\,(1 + bk)$$

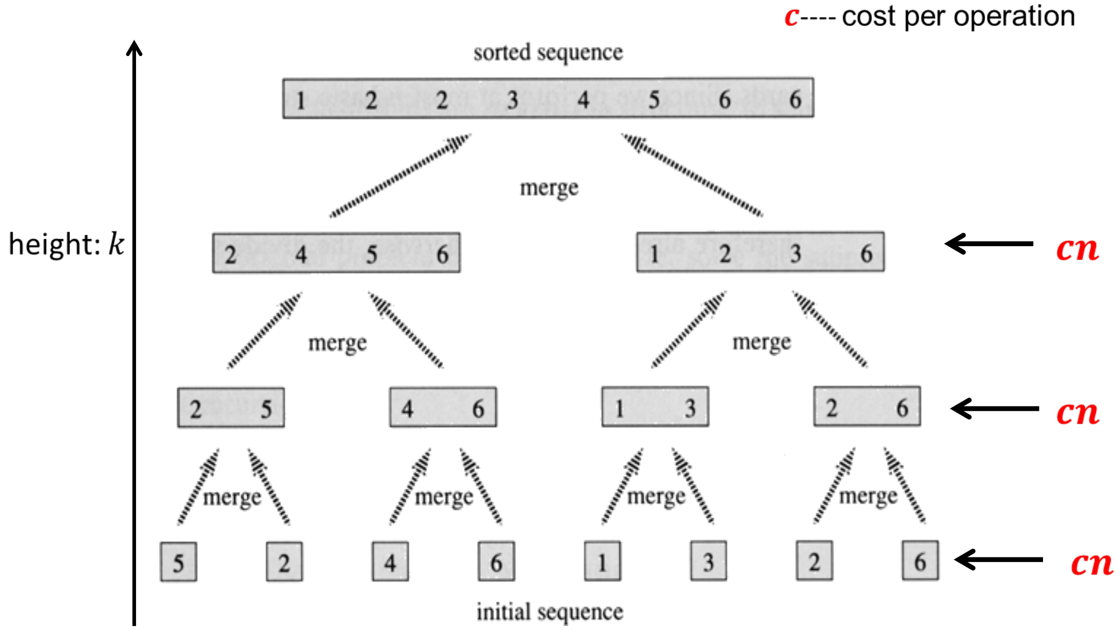
but $k = \log_2 n$, so

$$T(n) = n + b\, n\, log_2 n$$

$$\boxed{= \boldsymbol{O(n\; log\; n)}}$$

$\boldsymbol{n}$ --- number of elements

$\boldsymbol{c}$---- cost per operation

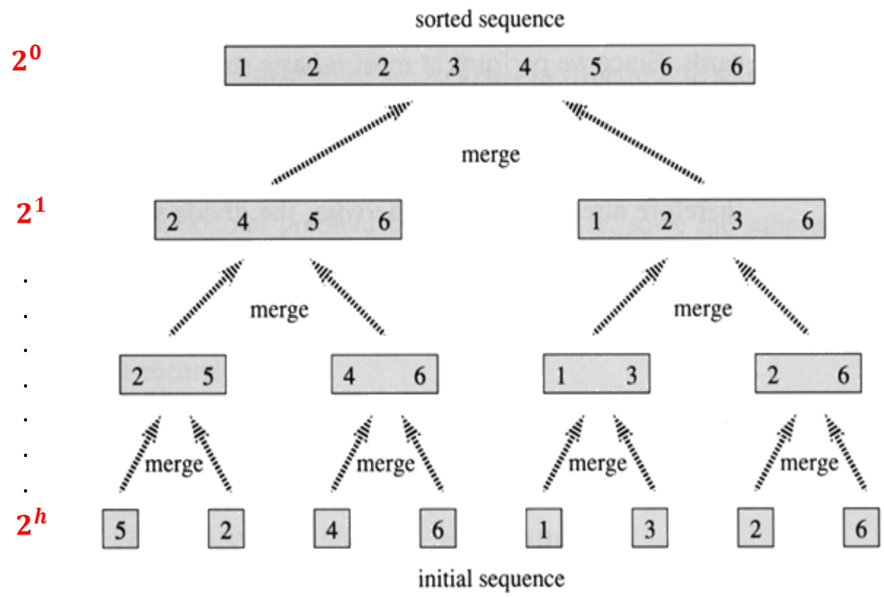


T(n) is $O(k \times \boldsymbol{cn})$

but $k = \log_2 n$ becuase $n = 2^k$

$T(n)\ is\ \boldsymbol{O(\log_2 n \times cn) = O(}\ n\ \mathrm{og}_2\ \boldsymbol{n}\ )$

assume $n = 2^h$
$\log_2 n = h$ $\Rightarrow$ $T(n) = \log_2 n \times cn$
$= cn \times \log_2 n$
$= O(n \log n)$



binary search: algorithm

function *binarySearch* (A, x, first, last)
    if first>last **return** −1
    middle=(first+last)/2
    if A[middle]=x **return** middle
    elseif A[middle]>x **return** binarySearch(A,x,first,middle−1)
    else return binarySearch(A,x,middle+1,last)

	0	1	2	3	4	5	6	7	8	9	10
A	34	54	123	330	431	556	567	587	888	988	999

first = 0
last = 10
x = 587

```
middle=(first+last)/2
If (A[middle]==x)
      return A[middle]
else if(A[middle]>x)
      last=middle-1;
else
      first=middle+1;
```

0	1	2	3	4	5	6	7	8	9	10
34	54	123	330	431	556	567	587	888	988	999

0	1	2	3	4	5	6	7	8	9	10
34	54	123	330	431	556	567	587	888	988	999

middle=(6+10)/2
= 8
A[8] > 587
last = 8-1 = 7
first = 6

0	1	2	3	4	5	6	7	8	9	10
34	54	123	330	431	556	567	587	888	988	999

binary search: complexity

$\boxed{\boldsymbol{T(n)\ is\ O(\log n)}}$

proof : $T(n)$ given by recurrence relation,

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

we consider only

$$T(n) = T(n/2) + O(1)$$

total search time

search time through half the list

each step involves one comparison

binary search: complexity

given

$T(n) = T(n/2) + O(1)$

expand, $T(n/2)$ i.e., put $T(n/2) = T((n/2)/2) + 1$

$= [T((n/2)/2) + 1] + 1$

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

expand $T(n/2^2)$

$= [T((n/2)/2^2) + 1] + 2$

$= T(n/2^3) + 3$

binary search: complexity

$T(n) = T(n/2^3) + 3$

⋯ by induction

$= T\left(n/2^i\right) + i$

as $n = 2^k$ for some k, i eventually become k

$= T\left(n/2^k\right) + k,$ put $2^k = n$

$T(n) = T(1) + k\,,$ as $T(1) = 1$

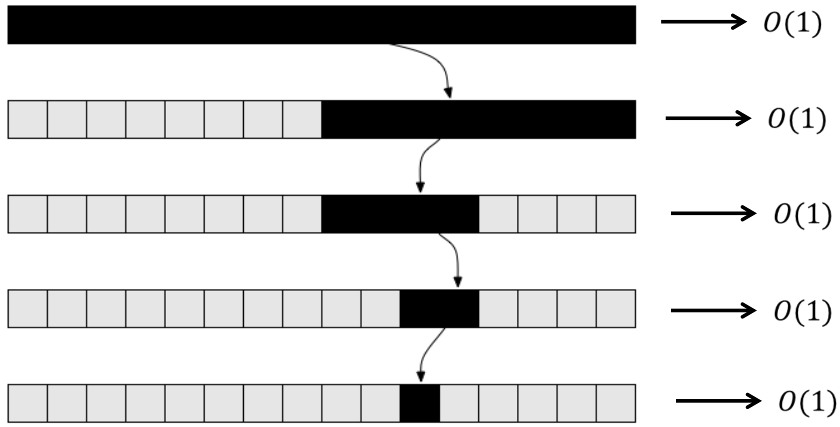
$= 1 + k\,,$

$2^k = 1$, solving for k
$k = \log_2 n$

$= 1 + \log_2 n\,,$

$\boldsymbol{T(n) = O(\log_2 n)}$,

binary search: complexity



$n = 2^k$

the implicit search tree has *k* levels

No of comparisons proportional to *k*

$T(n) = ck$ *because* $n = 2^k$, solving for k, $k = \log_2 n$

$$T(n) = c \log_2 n$$

End