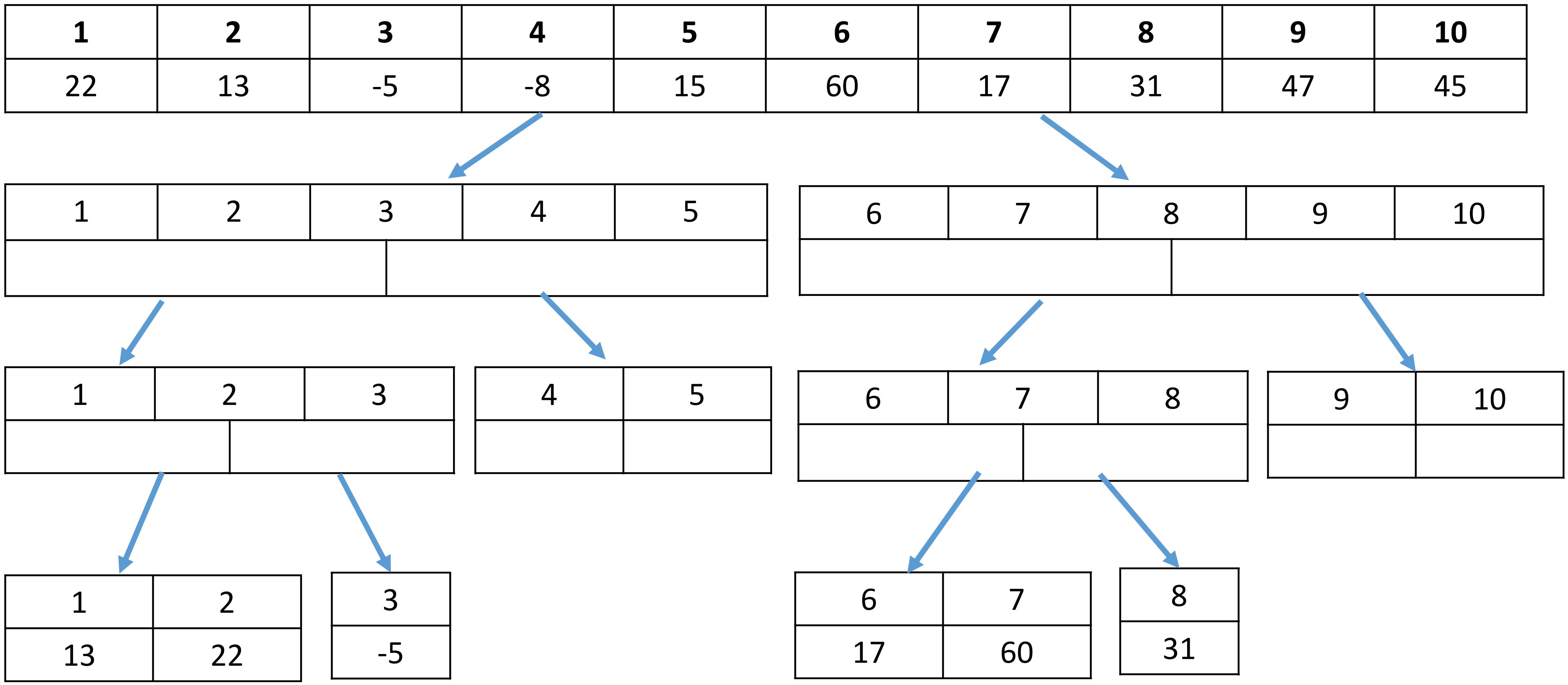
Week 4

hodcs@rknec.edu

Min-Max: DAC based Example



Min-Max: DAC Algorithm

```
Algorithm min_max(a,i,j,min,max)
{
If (i==j) then min=max=a[i]
Else
If(i=j-1) then
{
   if(a[i] < a[j]) then
   {
      min = a[i]; max=a[j]
   }
   else
   {
      min=a[j]; max=a[i]
   }
Else
  {
      mid = [i+j] / 2 (Take lower integer)
      min_max(a,i,mid,min,max)
      min_max(a,mid+1, j, min1, max1)
      if (max < max1) then max = max1
      if (min > min1) then min = min 1
  }
}
```

- Complexity equation:
- $T(n) = T(n/2) + T(n/2) + 2$ for $n > 2$
- $T(n) = 1$ if $n=2$
- $T(n) = 0$ if $n=1$

Quick Sort Analysis

```
Algorithm QuickSort(a, start, end)
{
    If (start < end)                                  C1
    {
        X = partition(a, start, end)          Check partition time
        Quicksort(a, start, x-1)                      T(n/2)
        Quicksort(a, x+1, end)                        T(n/2)
    }
}
```

Total time including partition = $2T(n/2) + an + b + c1 \Rightarrow 2T(n/2) + an$

Which can be written as: $2T(n/2) + cn$, where c = constant.

If there is only one element, then $T(1) = c$, because "if" loop will not be executed.

Quick Sort Analysis

```
Algorithm partition(a, start, end)
{
     pivot = a[end];                              1 unit
     pindex = start;                              1 unit
     for i = start to end-1 do
     {
         if (a[i] <= pivot)
         {
             swap(a[i], a[pindex])            "a" time unit for "n" times
             pindex++;
         }
     swap(a[pindex], pivot)                       1 unit
     return(pindex)                               1 unit
}
Total time = an+b
```

Mathematical solving

T(n) = 2T(n/2) + cn

In the next recursion, the value of n = n/4, i.e., n replaced by n/2.

T(n) = $2\left[2\left[T\left(\frac{n}{4}\right)+c[n/2]\right]\right]+cn$

T(n) = 4T(n/4) + 2 cn

T(n) = 8T(n/8) + 3cn

T(n) = $2^k\ T\left[\frac{n}{2^k}\right]+k\ cn$

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

T(n) = $2^{\log n}\ *\ 1+logn\ c\ n\ {=}{\Longrightarrow}\ n\log n$

Worst Case

- In worst case the given array is already sorted and it is required to sort it in reverse order. (ascending to descending).
- N

1	2	3	4	5	6	7	8

- And in all steps, the resultant arrays will be un-balanced.
- In such case: the second Quicksort function will not be executed and time complexity will be controlled by only first quicksort function. This function will be executed for $T(n-1)$ times.
- $T(n) = T(n-1) + cn$
- $T(n) = T(n-2) + c(n-1) + cn = T(n-2) + 2cn – c$
- $T(n) = T(n-3) + c(n-1) + 2cn – c$ ➔
- $T(n-3) + c(n-2) + 2cn – c = T(n-3) + 3cn – 3c$
- $T(n-4) + 4cn – 6c$
- $T(n-k) + kcn – (k(k-1)/2)$
- Smallest unit = $n-k= 1$, hence $k = n$ ignoring the constants.
- ***$T(n) = T(1) + ncn$ – constant term ➔ $nxn = n^2$***

Unit 4: Dynamic Programming

hodcs@rknec.edu

Characteristics

- **Developed by Richard Bellman in 1950.**
- **To provide accurate solution with the help of series of decisions.**
- **Define a small part of problem and find out the optimal solution to small part.**
- **Enlarge the small part to next version and again find optimal solution.**
- **Continue till problem is solved.**
- **Find the complete solution by collecting the solution of optimal problems in bottom up manner.**

Characteristics

- Types of problems and solution strategies.
- **1. Problem can be solved in stages.**
- **2. Each problem has number of states**
- **3. The decision at a stage updates the state at the stage into the state for next stage.**
- **4. Given the current state, the optimal decision for the remaining stages is independent of decisions made in previous states.**
- **5. There is recursive relationship between the value of decision at a stage and the value of optimum decisions at previous stages.**

Characteristics

- How memory requirement is reduce:
- How recursion overheads are converted into advantage
- - By storing the results of previous n-2 computations in computation of n-1 stage and will be used for computation of “n” stage.
- Not very fast, but very accurate
- It belongs to smart recursion class, in which recursion results and decisions are used for next level computation. Hence not only results but decisions are also generated.
- **Final collection of results: Bottom Up Approach.**

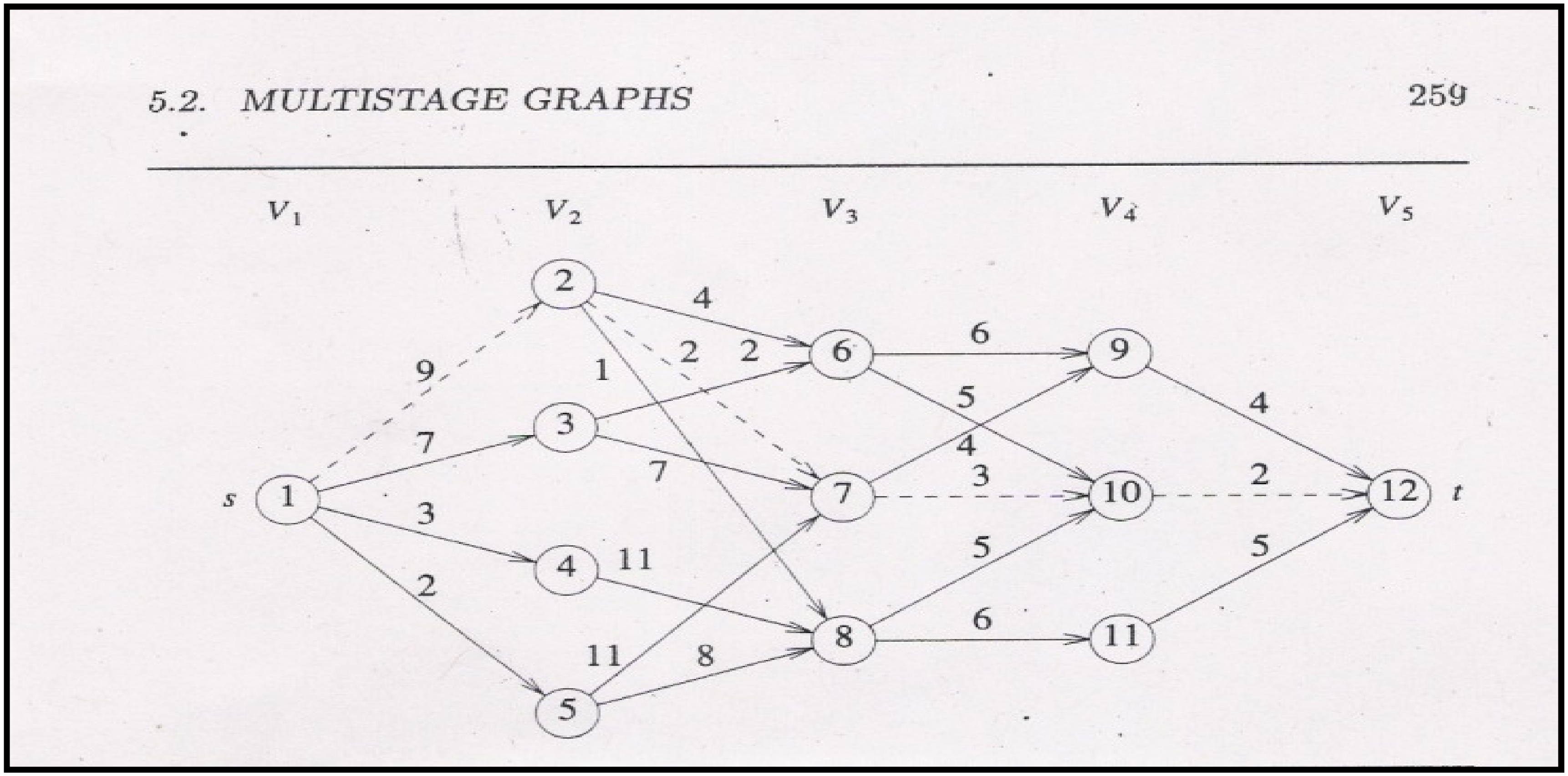
Definition

I) MULTI-STAGE GRAPH:

Features:

1) It is a directed graph consist of Vertices and Edges
2) In this graph the vertices are partitioned into $k >= 2$ disjoint sets. These sets are denoted as Vi, where $1 <= i <= k$
3) If there is an edge <u, v>, then the vertex "u" belongs to set "Vi" and vertex "v" belongs to set "V_{i+1}"
4) The number of vertices in the first and last set i.e., V_1 and V_k are 1, that is there will be only one source and one destination vertex.

Example of Multi-stage Graph



Backward Algorithm

Backward Algorithm:

Using this method, the shortest path from source to destination, is generated using backward movement. The starting stage of the graph is denoted as stage 2, so that the backward movement can be implemented to stage 1.

The algorithm uses formula:

$$bcost(i, j) = \min_{\substack{l \in V_{i-1} \\ \langle l, j \rangle \in E}} \{ bcost(i-1, l) + cost(l,j) \}$$

In this: "i" represent : STAGE *"j" represents: VERTICES OF STAGE*

i -1 represents: PREVIOUS STAGE *"l" represenst: VERTICES OF PREVIOUS STAGE*

The graph is represented using "cost" matrix of size "nxn".

Algorithm

Algorithm: Backward Method

Assumptions:

1) The graph is represented using cost matrix of size [nxn].
2) Array d[1..n] is a temporary array used to hold vertices information
3) Array p[1..k] is a output array of size "k", where "k" is number of stages in the graph.

```
Algorithm backward_cost(G, cost, d, n : p)
{
    Step 1:
        bcost[1] = 0 //Assume vertex 1 as start vertex
    Step 2:
        For j = 2 to n do
        {
        Function: Find vertex "r" such that <r,j> is an edge in the graph and bcost[r] +
cost[r,j] is minimum
                bcost[j] = bcost[r] + cost[r,j];
                d[j] = r;
        } //End of For loop
    Step 3: Finding minimum cost path
        p[1] = 1;        p[k] = n
        For j = k-1 to 2 do
                p[j] = d[p[j+1]];
} //End of Algorithm
```