Optimal read list for given number of days

A person is determined to finish the book in 'k' days but he never wants to stop a chapter in between. Find the optimal assignment of chapters, such that the person doesn't read too many extra/less pages overall.

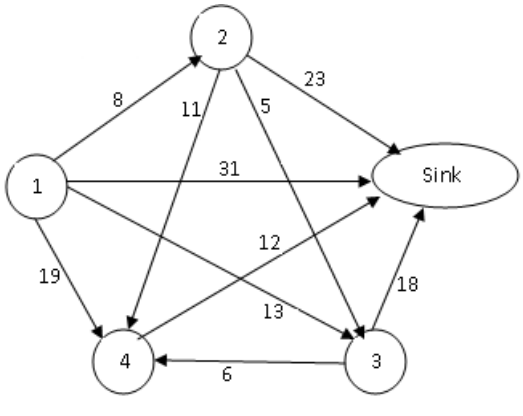
```
Example 1:
Input:  Number of Days to Finish book = 2
        Number of pages in chapters = {10, 5, 5}
Output: Day 1:  Chapter 1
        Day 2:  Chapters 2 and 3

Example 2:
Input:  Number of Days to Finish book = 3
        Number of pages in chapters = {8, 5, 6, 12}
Output: Day 1:  Chapter 1
        Day 2:  Chapters 2 and 3
        Day 2:  Chapter 4
```

The target is to minimize the sum of differences between the pages read on each day and average number of pages. If the average number of pages is a non-integer, then it should be rounded to closest integer.

In above example 2, average number of pages is (8 + 5 + 6 + 12)/3 = 31/3 which is rounded to 10. So the difference between average and number of pages on each day for the output shown above is "abs(8-10) + abs(5+6-10) + abs(12-10)" which is 5. The value 5 is the optimal value of sum of differences.

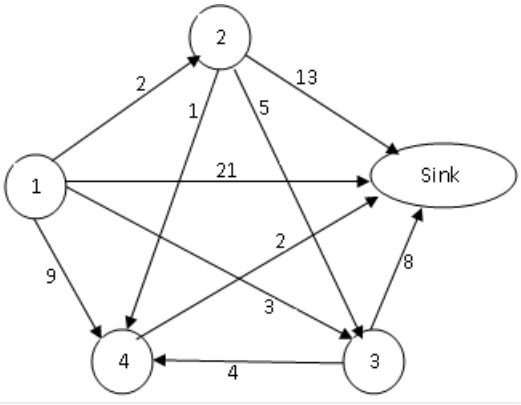
We strongly recommend to minimize the browser and try this yourself first.

Consider the example 2 above where a book has 4 chapters with pages 8, 5, 6 and 12. User wishes to finish it in 3 days. The graphical representation of the above scenario is,



In the above graph vertex represents the chapter and an edge e(u, v) represents number of pages to be read to reach 'v ' from 'u '. Sink node is added to symbolize the end of book.

First, calculate the average number of pages to read in a day (here 31/3 roughly 10). New edge weight e '(u, v) would be the mean difference |avg – e(u, v)|. Modified graph for the above problem would be,



Thanks to Armadillo for initiating this thought in a comment.

The idea is to start from chapter 1 and do a DFS to find sink with count of edges being 'k '. Keep storing the visited vertices in an array say 'path[]'. If we reach the destination vertex, and path sum is less than the optimal path update the optimal assignment optimal_path[]. Note, that the graph is DAG thus there is no need to take care of cycles during DFS.
Following, is the C++ implementation of the same, adjacency matrix is used to represent the graph. The following program has mainly 4 phases.
1) Construct a directed acyclic graph.
2) Find the optimal path using DFS.
3) Print the found optimal path.

```
// C++ DFS solution to schedule chapters for reading in
// given days
# include <iostream>
# include <cstdlib>
# include <climits>
# include <cmath>
using namespace std;

// Define total chapters in the book
// Number of days user can spend on reading
# define CHAPTERS 4
# define DAYS 3
# define NOLINK -1

// Array to store the final balanced schedule
int optimal_path[DAYS+1];

// Graph - Node chapter+1 is the sink described in the
//         above graph
int DAG[CHAPTERS+1][CHAPTERS+1];

// Updates the optimal assignment with current assignment
void updateAssignment(int* path, int path_len);

// A DFS based recursive function to store the optimal path
// in path[] of size path_len.  The variable sum stores sum of
// of all edges on current path.  k is number of days spent so
// far.
void assignChapters(int u, int* path, int path_len, int sum, int k)
{
    static int min = INT_MAX;

    // Ignore the assignment which requires more than required days
    if (k < 0)
        return;

    // Current assignment of chapters to days
    path[path_len] = u;
    path_len++;

    // Update the optimal assignment if necessary
    if (k == 0 && u == CHAPTERS)
    {
        if (sum < min)
        {
            updateAssignment(path, path_len);
```

```
            updateAssignment(path, path_len);
            min = sum;
        }
    }

    // DFS - Depth First Search for sink
    for (int v = u+1; v <= CHAPTERS; v++)
    {
        sum += DAG[u][v];
        assignChapters(v, path, path_len, sum, k-1);
        sum -= DAG[u][v];
    }
}

// This function finds and prints optimal read list.  It first creates a
// graph, then calls assignChapters().
void minAssignment(int pages[])
{
    // 1) ............CONSTRUCT GRAPH.................
    // Partial sum array construction S[i] = total pages
    // till ith chapter
    int avg_pages = 0, sum = 0, S[CHAPTERS+1], path[DAYS+1];
    S[0] = 0;

    for (int i = 0; i < CHAPTERS; i++)
    {
        sum += pages[i];
        S[i+1] = sum;
    }

    // Average pages to be read in a day
    avg_pages = round(sum/DAYS);

    /* DAG construction vertices being chapter name &
     * Edge weight being |avg_pages - pages in a chapter|
     * Adjacency matrix representation  */
    for (int i = 0; i <= CHAPTERS; i++)
    {
        for (int j = 0; j <= CHAPTERS; j++)
        {
            if (j <= i)
                DAG[i][j] = NOLINK;
            else
            {
                sum = abs(avg_pages - (S[j] - S[i]));
                DAG[i][j] = sum;
            }
        }
    }

    // 2) ............FIND OPTIMAL PATH.............
    assignChapters(0, path, 0, 0, DAYS);

    // 3) ..PRINT OPTIMAL READ LIST USING OPTIMAL PATH....
    cout << "Optimal Chapter Assignment :" << endl;
    int ch;
    for (int i = 0; i < DAYS; i++)
    {
        ch = optimal_path[i];
        cout << "Day" << i+1 << ": " << ch << " ";
        ch++;
        while ( (i < DAYS-1 && ch < optimal_path[i+1]) ||
                (i == DAYS-1 && ch <= CHAPTERS))
        {
           cout <<  ch << " ";
           ch++;
        }
        cout << endl;
    }
}

// This funtion updates optimal_path[]
void updateAssignment(int* path, int path_len)
```

```
{
    for (int i = 0; i < path_len; i++)
        optimal_path[i] = path[i] + 1;
}

// Driver program to test the schedule
int main(void)
{
    int pages[CHAPTERS] = {7, 5, 6, 12};

    // Get read list for given days
    minAssignment(pages);

    return 0;
}
```

Output:

```
Optimal Chapter Assignment :
Day1: 1
Day2: 2 3
Day3: 4
```