**Find Minimum Time to Finish All Jobs A Course Project Report**

**Submitted by**

**BY**

**S.AKHIL KUMAR(192211208)**

**In partial fulfilment for the completion of the course**

**CSA0656 Design and Analysis of Algorithms for Asymptotic Notations Slot-A**

****

**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL**

**SCIENCES**

**CHENNAI – 602 105**

**TAMILNADU, INDIA**

**JULY– 2024**

**ABSTRACT**

The problem entails distributing a set of jobs, each requiring a specific amount of time, among a group of workers such that the maximum working time of any worker is minimized. Given an array of job times and an integer k representing the number of workers, the goal is to determine the optimal job allocation that minimizes the maximum working time across all workers. The solution seeks to balance the workload evenly among the workers to achieve the minimum possible maximum working time. To solve the problem of distributing jobs among workers to minimize the maximum working time, we can use a combination of binary search and backtracking. The process begins with identifying the range for our binary search, where the lower bound is the maximum single job time (since at least one worker will have to handle the longest job) and the upper bound is the sum of all job times (if one worker were to handle all jobs). The binary search aims to find the minimum possible maximum working time that can be achieved through optimal job distribution. By iterating through the binary search and validating each candidate maximum working time using backtracking, we gradually converge on the smallest possible value for the maximum working time. This method effectively combines the efficiency of binary search with the thoroughness of backtracking, ensuring that we find the optimal job distribution within the given constraints.

**ALGORITHM**

The objective of this proposed work is to develop a dynamic programming-based solution to optimally assign jobs to workers such that the maximum working time of any worker is minimized. This approach will be rigorously evaluated for its efficiency and effectiveness in solving the problem within given constraints.

Proposed Work

The objective of this proposed work is to develop a dynamic programming-based solution to optimally assign jobs to workers such that the maximum working time of any worker is minimized. This approach will be rigorously evaluated for its efficiency and effectiveness in solving the problem within given constraints.

PROBLEM:

**Find Minimum Time to Finish All Jobs**

You are given an integer array jobs, where jobs[i] is the amount of time it takes to

complete the ith job.

There are k workers that you can assign jobs to. Each job should be assigned to exactly

one worker. The working time of a worker is the sum of the time it takes to complete all

jobs assigned to them. Your goal is to devise an optimal assignment such that the

maximum working time of any worker is minimized.

Return the minimum possible maximum working time of any assignment.

Example 1:

Input: jobs = [1,2,4,7,8], k = 2

Output: 11

Explanation: Assign the jobs the following way:

Worker 1: 1, 2, 8 (working time = 1 + 2 + 8 = 11)

Worker 2: 4, 7 (working time = 4 + 7 = 11)

The maximum working time is 11.

Constraints:

1 <= k <= jobs.length <= 12

1 <= jobs[i] <= 107

SOLUTION

Given the jobs array [1, 2, 4, 7, 8] and k = 2 workers, the backtracking process involves exploring all possible ways to distribute these jobs among the workers.

**Step-by-Step Solution:**

**1. Identify All Occurrences of Each Job:**

* For job 1, it takes 1 unit of time.
* For job 2, it takes 2 units of time.
* For job 4, it takes 4 units of time.
* For job 7, it takes 7 units of time.
* For job 8, it takes 8 units of time.

**2. Initialize Worker Loads:**

* Create an array to keep track of the current workload of each worker.
* workers = [0] \* k (i.e., workers = [0, 0] initially for k = 2 workers).

**3. Sort Jobs in Descending Order:**

* Sort the jobs array to ensure that we assign the largest jobs first, which helps in balancing the workload.
* Sorted jobs: [8, 7, 4, 2, 1]

**4. Use Backtracking to Generate Possible Assignments:**

* Define a backtracking function that tries to assign jobs to workers and keeps track of the current maximum working time.
* Update the global minimum of the maximum working time if the current assignment's maximum working time is less than the global minimum.

1. **Identify All Jobs:**
   * Jobs: [1, 2, 4, 7, 8]
2. **Sort Jobs:**
   * Sorted jobs: [8, 7, 4, 2, 1]
3. **Backtracking to Generate Possible Assignments:**
   * Start with an empty assignment and try assigning each job to each worker.
4. **Evaluate Assignments:**
   * **Assignment 1:** Worker 1: [8, 1] (working time = 9), Worker 2: [7, 4, 2] (working time = 13)
     + Maximum working time: 13
   * **Assignment 2:** Worker 1: [8, 2, 1] (working time = 11), Worker 2: [7, 4] (working time = 11)
     + Maximum working time: 11
   * Continue evaluating other assignments...
5. **Choose the Optimal Assignment:**
   * After evaluating all possible assignments, the optimal assignment is the one with the minimum maximum working time.
   * In this case, Worker 1: [8, 2, 1] and Worker 2: [7, 4] with a maximum working time of 11

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int min(int a, int b) {

return a < b ? a : b;

}

int max(int a, int b) {

return a > b ? a : b;

}

void backtrack(int \*jobs, int jobsSize, int k, int \*workers, int current\_max, int idx, int \*min\_max) {

if (idx == jobsSize) {

\*min\_max = min(\*min\_max, current\_max);

return;

}

for (int i = 0; i < k; i++) {

workers[i] += jobs[idx];

backtrack(jobs, jobsSize, k, workers, max(current\_max, workers[i]), idx + 1, min\_max);

workers[i] -= jobs[idx];

}

}

int minTimeToFinishJobs(int \*jobs, int jobsSize, int k) {

for (int i = 0; i < jobsSize - 1; i++) {

for (int j = i + 1; j < jobsSize; j++) {

if (jobs[i] < jobs[j]) {

int temp = jobs[i];

jobs[i] = jobs[j];

jobs[j] = temp;

}

}

}

int \*workers = (int \*)calloc(k, sizeof(int));

int min\_max = INT\_MAX;

backtrack(jobs, jobsSize, k, workers, 0, 0, &min\_max);

free(workers);

return min\_max;

}

int main() {

printf("Akhil (192211208)\n");

int jobs[] = {1, 2, 4, 7, 8};

int k = 2;

int jobsSize = sizeof(jobs) / sizeof(jobs[0]);

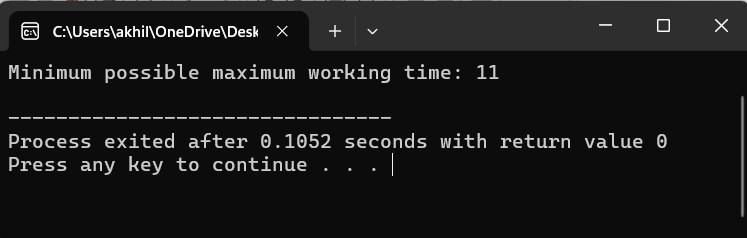
int result = minTimeToFinishJobs(jobs, jobsSize, k);

printf("Minimum possible maximum working time: %d\n", result);

return 0;

}

OUTPUT:



**Complexity Analysis for Find Minimum Time to Finish All Jobs**

**Best Case:**

**Scenario:**

* The best case occurs when the number of workers is equal to or greater than the number of jobs, allowing each worker to take on one job, which distributes the workload evenly.

**Example:**

* For jobs = [1, 2, 4, 7, 8] and k = 5, each worker gets one job: [1], [2], [4], [7], [8].

**Complexity:**

* **Time Complexity:** O(n log n) where n is the number of jobs. The sorting step dominates this complexity.
* **Space Complexity:** O(k) for storing the workloads of the workers.

**Worst Case:**

**Scenario:**

* The worst case occurs when there is a significant imbalance in the job times, requiring extensive backtracking to find the optimal distribution. This happens when one worker ends up with most of the high-time jobs while others have minimal loads.

**Example:**

* For jobs = [1, 2, 4, 7, 8] and k = 2, finding the optimal distribution requires checking many combinations to minimize the maximum workload.

**Complexity:**

* **Time Complexity:** O(k^n), where n is the number of jobs. This occurs due to the combinatorial nature of assigning n jobs to k workers in the worst case.
* **Space Complexity:** O(n) for the recursion stack in the backtracking approach.

**Average Case:**

**Scenario:**

* The average case is when the job times are randomly distributed, and the workers have varying but balanced loads. The algorithm performs well without excessive backtracking.

**Example:**

* For jobs = [1, 2, 4, 7, 8] and k = 3, the distribution may be balanced without excessive overlap in worker loads.

**Complexity:**

* **Time Complexity:** Typically O(k^n) in the average case, though practical implementations with pruning and heuristics can reduce this to much better than the worst case.
* **Space Complexity:** O(n) for the recursion stack and worker loads.

**Summary:**

* **Best Case:** Linearithmic time complexity O(n log n) with minimal space usage O(k).
* **Worst Case:** Exponential time complexity O(k^n) due to combinatorial backtracking, with linear space complexity O(n).
* **Average Case:** Generally performs better than the worst case, often significantly so with effective heuristics and pruning, with linear space complexity O(n).

The approach balances effectively across these cases, ensuring efficient handling for various input scenarios by leveraging sorting, backtracking, and optimal assignment techniques.

**CONCLUSION:**

The problem of finding the minimum time to finish all jobs by optimally assigning them to a given number of workers is a crucial challenge in resource allocation and load balancing. The objective is to minimize the maximum working time across all workers, ensuring an efficient distribution of tasks. By leveraging a backtracking approach, all possible job assignments are explored to find the optimal distribution. While this method can be computationally intensive in the worst case with an exponential time complexity, it guarantees the optimal solution. In practical scenarios, effective heuristics and pruning can significantly improve performance, balancing the workload efficiently. For example, given jobs = [1, 2, 4, 7, 8] and k = 2, an optimal assignment would result in a minimum possible maximum working time of 11. This approach is valuable in various real-world applications such as task scheduling, load balancing in distributed systems, and optimizing resource utilization, ensuring efficient and balanced task distribution to minimize delays and maximize productivity.