Ad-Words Problem

By:

Gautam Mittal **(21UCS181)**

Jeetaksh Gandhi **(21UCS098)**

Mukul Verma **(21UCS134)**

ACKNOWLEDGEMENT

We would like to express our sincerest gratitude to our Mining of Massive Datasets instructor **Dr. Subrat K. Dash** for giving us this wonderful opportunity to work together as a team.

Working on this project helped us all to develop a deeper and practical understanding of the subject. Moreover, it helped us in getting a glimpse of the upcoming professional life by learning to collaborate and encompassing different ideas.

DISCLAIMER

The project initially chosen by us: "**Implementation of SON Algorithm**" turned out to be too ambitious for us to complete in a weeklong time-frame. As absolute beginners, Hadoop seemed complex for us to understand in the limited time.

But we didn't stop there, we even tried to explore Apache Spark in Python to create the Distributed Environment for Map-Reduce simulation, but considering the time constraints, we had to ultimately move onto a different problem statement altogether in order to finish on time.

Hence, we have decided to cover the topic:

"AdWords and Generalized Matching Problem"; majorly focusing on the Balance Algorithm and its implementation.

INTRODUCTION

The AdWords problem is a fundamental optimization challenge in the realm of online advertising, particularly pertinent to search engine marketing platforms such as Google AdWords.

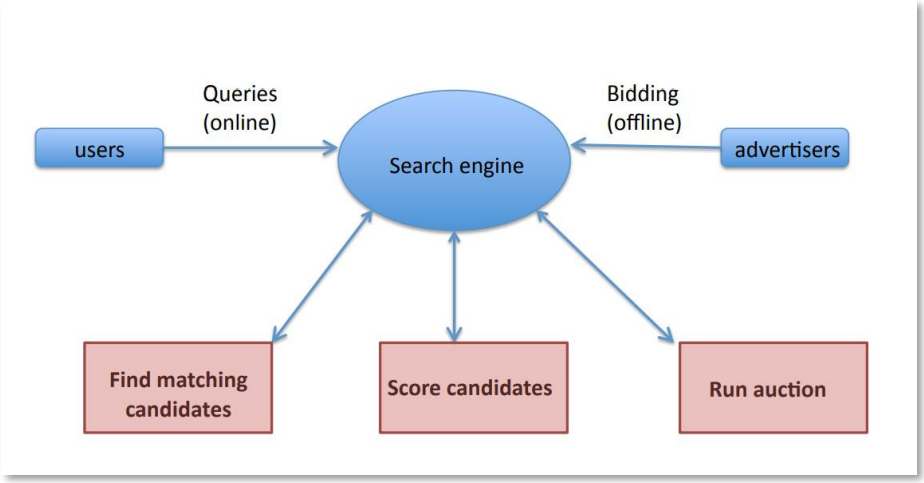
This problem centers on the allocation of advertisement slots to advertisers in response to user queries, with the objective of maximizing overall revenue while adhering to predefined budget constraints.

WHAT EXACTLY IS **ADWORD**?

The AdWords is essentially a **large auction** where businesses place bids for individual keywords, together with limits specifying their maximum daily budget.

The search engine company earns revenue from businesses when it displays their ads in response to a relevant search query (*if the user clicks on the ad*).

Indeed, most of the revenues of search engine companies are derived in this manner.



To summarize it in one line:

Assign user queries to advertisers to maximize the total revenue.

The Simplest Solution to Ad-Word Problem:

Use **raw bids** for each search query. [Initial strategy introduced by Overture]

*However, i*nstead of raw bids, Google identified that using the "expected revenue per click" (i.e., Bid*CTR) proves to be more profitable, let us demonstrate this with an example:

Advertiser's Bid's:

- A bids $1 for each click. (Click Through Rate – 2.5%)
- B bids $3 for each click. (Click Through Rate – 1%)
- C bids $5 for each click. (Click Through Rate – 0.25%

CASE 1: Overture's Initial Strategy

Advertiser	Bid
A	$1.00
B	$0.75
C	$0.50

This strategy, although puts forward the advertiser A, doesn't guarantee that the revenue would be as expected.

CASE 2: Google's "**Ad-Word**" Strategy

Advertiser	Bid	CTR	Bid * CTR
A	$1.00	1%	1 cent
B	$0.75	2%	1.5 cents
C	$0.50	2.5%	1.125 cents

Here, the **Expected Revenue** is the product of **Bid** with **Click Through Rate (CTR)**, Which can provide a better ordering of Advertisers if used to sort the table.

Advertiser	Bid	CTR	Bid * CTR
B	$0.75	2%	1.5 cents
C	$0.50	2.5%	1.125 cents
A	$1.00	1%	1 cent

[Same table after being sorted in the decreasing order of **Expected Revenue**]

Although there's no guarantee here either, the expected revenue is better than the unjustified faith in case of maximum bid value ordering.

Conclusion:

Performance based Algorithm helps us to maximize the revenue by taking into consideration the performance in the form of **CTR (Click - Through Rate)** and **Bid price**.

Hence priority will be given to B and C irrespective of A paying higher for a click to maximize the revenue.

DOES THIS NOT SOLVE THE ADWORD PROBLEM?

Although this feels like the solution of Ad-Word Problem, it isn't a viable due to the following complications:

- **Budget Constraints -** Each advertiser has a Limited budget.
- **CTR of advertisements is not known in advance**, as is measured historically.

WHY NOT RELY ON **CTR** COMPLETELY?

Since CTR can't be known in advance, we try to improve the allocation strategy using the known data, and hence we move onto our first strategy of allocation; the **Greedy Allocation**.

Algorithm 8: GREEDY For Adwords

When the next vertex $v \in V$ **arrives:**
If all neighbors of v are unavailable (that is, have spent their budgets), continue.
Else match v to that available neighbor u which has the maximum value of bid_{uv}.

To understand Greedy, let us consider an example:

Consider the following simplified environment.

♣ There is 1 ad shown for each query.

♣ All advertisers have the same budget **B.**

♣ All ads are equally likely to be clicked.

♣ Value of each ad is the same (=1)

EXAMPLE – 1: OFFLINE Greedy Allocation

Suppose we have 2 advertisers **A** and **B**:

♣ **A** bids on query x & y B bids on x
♣ Both have budgets of $4.
ϒ **Query stream: x x x x y y y y**

♣ Solution of this AdWords Problem could generate the worst-case greedy choice as:

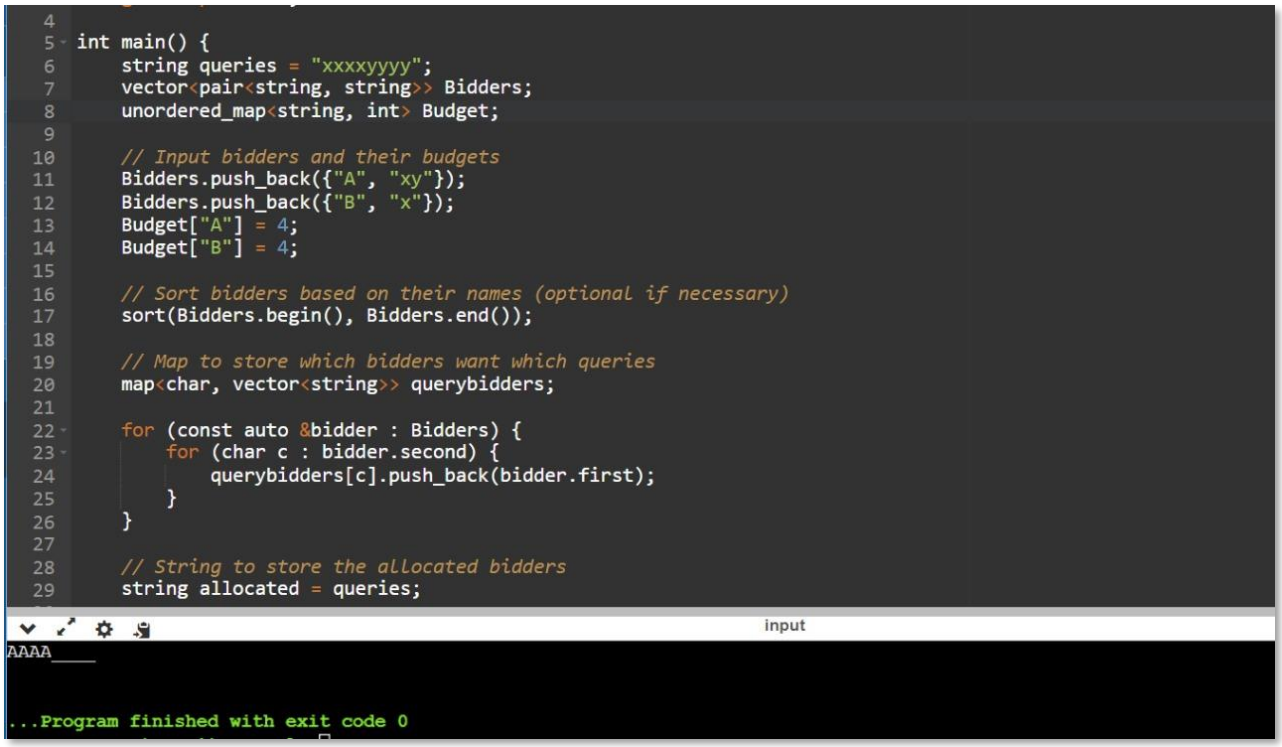
A A A A _ _ _ _

♣ Whereas the optimal allocation would have been **B B B B A A A A**

IMPLEMENTATION:

The greedy algorithm has also been implemented by us, the code of which has also been attached along with this report.

Here's the overview of the implementation:



```
int main() {
    string queries = "xxxxyyyy";
    vector<pair<string, string>> Bidders;
    unordered_map<string, int> Budget;

    // Input bidders and their budgets
    Bidders.push_back({"A", "xy"});
    Bidders.push_back({"B", "x"});
    Budget["A"] = 4;
    Budget["B"] = 4;

    // Sort bidders based on their names (optional if necessary)
    sort(Bidders.begin(), Bidders.end());

    // Map to store which bidders want which queries
    map<char, vector<string>> querybidders;

    for (const auto &bidder : Bidders) {
        for (char c : bidder.second) {
            querybidders[c].push_back(bidder.first);
        }
    }

    // String to store the allocated bidders
    string allocated = queries;
```

```
input
AAAA____

...Program finished with exit code 0
```

As we can see, the GREEDY Algorithm isn't the most efficient when it comes to solving Ad-Word problem.

As a matter of fact,

Competitive ratio of greedy is 1/2.

But this is the allocation which we achieve when we know the query stream as well as advertisers **in advance.**

When decision needs to be taken ONLINE, **without delay** ⟶ Online Algorithms

So, we also have an implementation of the Online Greedy Algorithm, which allocates the query to the advertiser following the greedy allocation strategy.

Here's the implementation:

IMPLEMENTATION:

main.cpp

```
67          // Check each bidder who is interested in the current query character
68          for (const auto &bidder : queryBidders[queryChar]) {
69              if (budget[bidder] > 0) {
70                  budget[bidder]--; // Deduct one unit from the bidder's budget
71                  allocatedQueries += bidder[0]; // Assign the query to this bidder
72                  isAllocated = true;
73                  break; // Stop after assigning to one valid bidder
74              }
75          }
76
77          // If no bidder could be allocated (budget exhausted), mark the query with '_'
78          if (!isAllocated) {
79              allocatedQueries += '_';
80          }
81      }
82      cout << allocatedQueries << endl; // Output the current allocation of queries
83
84      budgetAvailable = hasRemainingBudget(budget); // Check if any budget is still available
85      if (!budgetAvailable) {
86          cout << endl << "All budgets exhausted " << endl;
```

input

```
Enter Bidder's budget:
5
Enter the queries one character at a time, enter $ to exit:
x
A
y
AB
x
ABA
x
ABAA
y
ABAAB
```

Even in the **ONLINE variant of Greedy**, the following theorem still holds:

Theorem 5.1. GREEDY achieves a ratio of $\frac{1}{2}$ for the Adwords problem (even without the small bids assumption).

CAN **GREEDY** BE IMPROVED?

One such attempt is the **Balance Algorithm.**

- Slight improvement over Greedy algorithm.
- Assigns query to advertiser who bids on the query & has the **largest remaining budget**.
- Ties are broken arbitrarily. **(but in a deterministic way)**

To demonstrate the working of Balance Algorithm, we once again consider the simplified environment example, and see the difference:

EXAMPLE – 2: OFFLINE BALANCE Algorithm

Suppose we have 2 advertisers **A** and **B**:

♣ **A** bids on query x & y B bids on x
♣ Both have budgets of $4.
ϒ **Query stream: x x x x y y y y**

♣ Following the improvement, BALANCE would produce: **A B A B B B _ _**
♣ Optimal: **A A A A B B B B**

In general: BALANCE for 2 advertisers has a

Competitive ratio = ¾

We have also implemented Offline Balance, which can be found in the project shared.

Here's the implementation overview:

```cpp
//Balance Offline
#include <bits/stdc++.h>
using namespace std;

// Function to calculate the Psi value based on bid and remaining budget fraction
double calculatePsi(double bid, double remainingFraction) {
    return bid * (1 - exp(-remainingFraction));
}

// Function to check if any bidder still has budget remaining
bool hasRemainingBudget(unordered_map<string, double> &budget) {
    for (auto &entry : budget) {
        if (entry.second > 0) {
            return true;
        }
    }
    return false;
}
```

input

```
Best Bidder: A with remaining budget 100
Allocated so far: A
Best Bidder: A with remaining budget 90
Allocated so far: AA
Best Bidder: A with remaining budget 80
Allocated so far: AAA
Best Bidder: A with remaining budget 70
Allocated so far: AAAA
Best Bidder: A with remaining budget 60
Allocated so far: AAAAA
Best Bidder: A with remaining budget 50
Allocated so far: AAAAAA
Best Bidder: A with remaining budget 40
Allocated so far: AAAAAAA
```

Again, the allocation is not useful unless it is ONLINE, so we also must consider ONLINE Balance algorithm.

The Pseudo-Code is as follows:

Algorithm 9: BALANCE

When the next vertex $v \in V$ arrives:

If all neighbors of v are unavailable (that is, have spent their budgets), continue.

Else match v to that available neighbor u which has spent the least fraction of its budget so far.

Following the above algorithm, here's the implementation for the Online Balance:

main.cpp

```cpp
//Balance Online
#include <bits/stdc++.h>
using namespace std;

// Function to check if any bidder still has budget remaining
bool hasRemainingBudget(unordered_map<string, int> &budget) {
    for (auto &entry : budget) {
        if (entry.second > 0) {
            return true;
        }
    }
    return false;
}

int main() {
    vector<pair<string, string>> bidders; // Vector to store bidders and their queries
    unordered_map<string, int> budget; // Map to store bidders and their budgets

    // Input number of bidders
    cout << "Enter the number of bidders:" << endl;
```

input

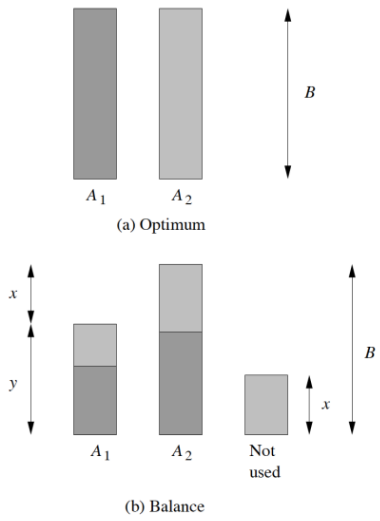
```
Enter the number of bidders:
2
Enter Bidder name:
A
Enter Bidder's queries:
q
Enter Bidder's budget:
110
Enter Bidder name:
B
Enter Bidder's queries:
q
Enter Bidder's budget:
100
```

Balance Algorithm

By theoretical proof, we can infer that competitive ratio of the Balance Algorithm is at least 3/4 (0.75) for the simple case of two advertisers.

This is an improvement over the Greedy allocation,

However, would that be the case when we have **more than 2 advertisers?**



BALANCE ALGORITHM WITH MANY BIDDERS

Also used to find out the general result of the Balance Algorithm; no other online algorithm for the same task has a better competitive ratio.

Worst Competitive Ratio of BALANCE is *1–1/e = approx. 0.63*

To illustrate this competitive ratio, let us consider the worst-case example that gives this ratio:

***N* advertisers: $A_1, A_2, \ldots A_N$**

- Each with budget $B > N$

Queries:

- $N \cdot B$ queries appear in N rounds of B queries each.

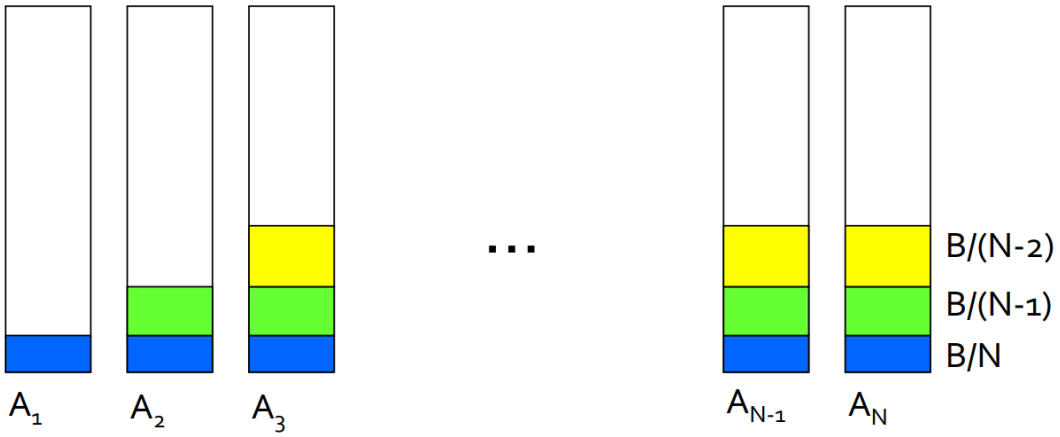
Bidding:

- Round **1** queries: bidders $A_1, A_2, \ldots, A_N$
- Round **2** queries: bidders $A_2, A_3, \ldots, A_N$
- Round i queries: bidders $A_i, \ldots, A_N$

Optimum allocation:

Allocate round i queries to A_i

- Optimum revenue $N \cdot B$



Balance Algorithm

- ...
- $A_1, \ldots, A_N$ get $B/(N - i + 1)$ of queries q_i
- ...
- Eventually, budgets of higher-numbered advertisers will be exhausted

After theoretical derivation, we figure out that after exactly N(1-1/e) rounds of allocation, we cannot allocate further queries to any of the advertisers.

After the first **k=N(1-1/e)** rounds, we cannot allocate a query to any advertiser.

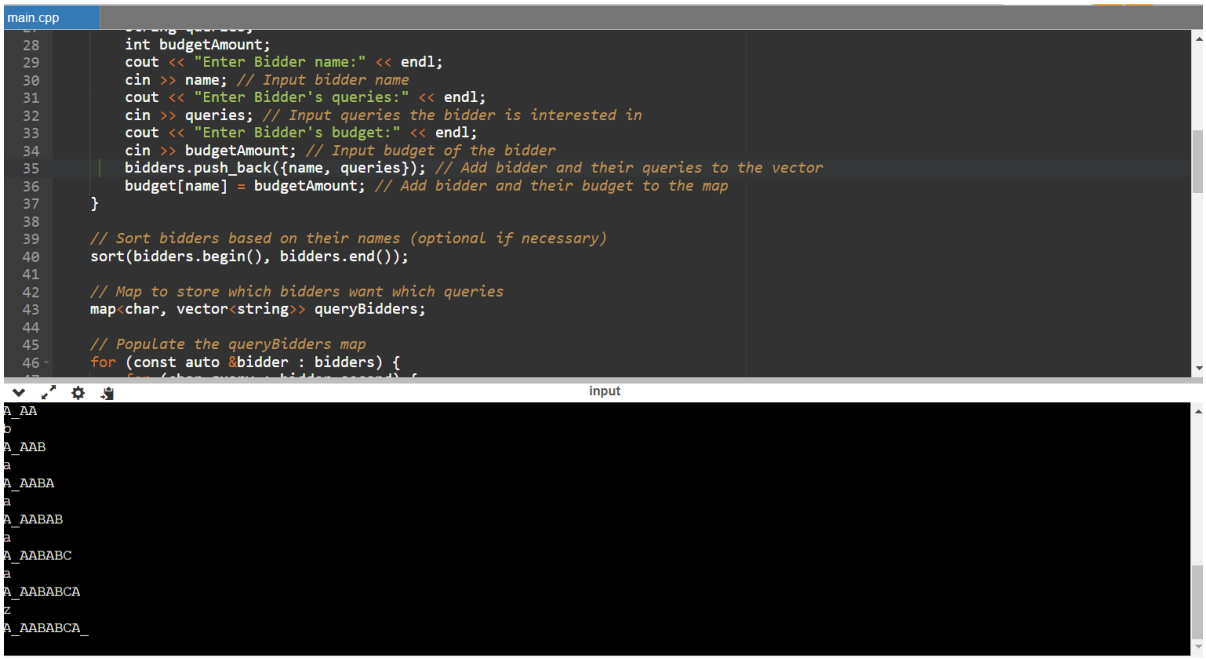
Hence,

- **Revenue Generated = B·N (1-1/e)**
- **Optimum Revenue N·B**

Competitive ratio = 1-1/e

We have implemented the algorithm for multiple bidders as well, the code of which has been shared.

IMPLEMENTATION:



So far, we have been considering a simplified environment for the sake of simplicity.

However, now that we have got the hang of the Ad-words problem, we can finally lift all the restrictions, and consider the **generalized case of the Ad-Words problem**.

SHORT-COMING OF BALANCE ALGORITHM

The balance algorithm obviously works wonder when the bids are binary (0 or 1).

However, in practice, bids are **arbitrary**, and with arbitrary bids.

In a general setting, Balance fails to deal with the arbitrary bids and arbitrary budgets,

We can observe the short-coming of the Balance algorithm through this simple example:

EXAMPLE – 3: Short-Coming of Balance Algorithm

Consider two advertisers **A1** and **A2**

A1: Bid for Query q = 1, Budget = 110
A2: Bid for Query q = 10, Budget = 100

Consider we see 10 instances of Query "q".

In this case, the Balance algorithm studied so far selects A1 and generates a revenue of 10$.

But an observant eye can identify that the **OPTIMAL** revenue would be **100$.**

THE GENERALIZED BALANCE ALGORITHM

In order to deal with the Arbitrary bids, we define a function $\Psi_i = x_i(1 - e^{-f_i})$.

Then we assign query 'q' to the advertiser A_i for which Ψ_i **is maximum**.

The following generalization of the Balance algorithm can be shown to have a competitive ratio of $1 - \frac{1}{e} \approx 0.63$:

Generalized Balance Algorithm

- Query q arrives
- Advertiser A_i has bid x_i for query q
- Advertiser A_i has fraction f_i of his budget left unspent
- Let

$$\Psi_i = x_i(1 - e^{-f_i}) \tag{7}$$

Then assign q to advertiser A_i such that Ψ_i is maximum.

We have followed the above generalization, and done the implementation:

Here's the implementation for the **OFFLINE** version of the Generalized Balance Algorithm:

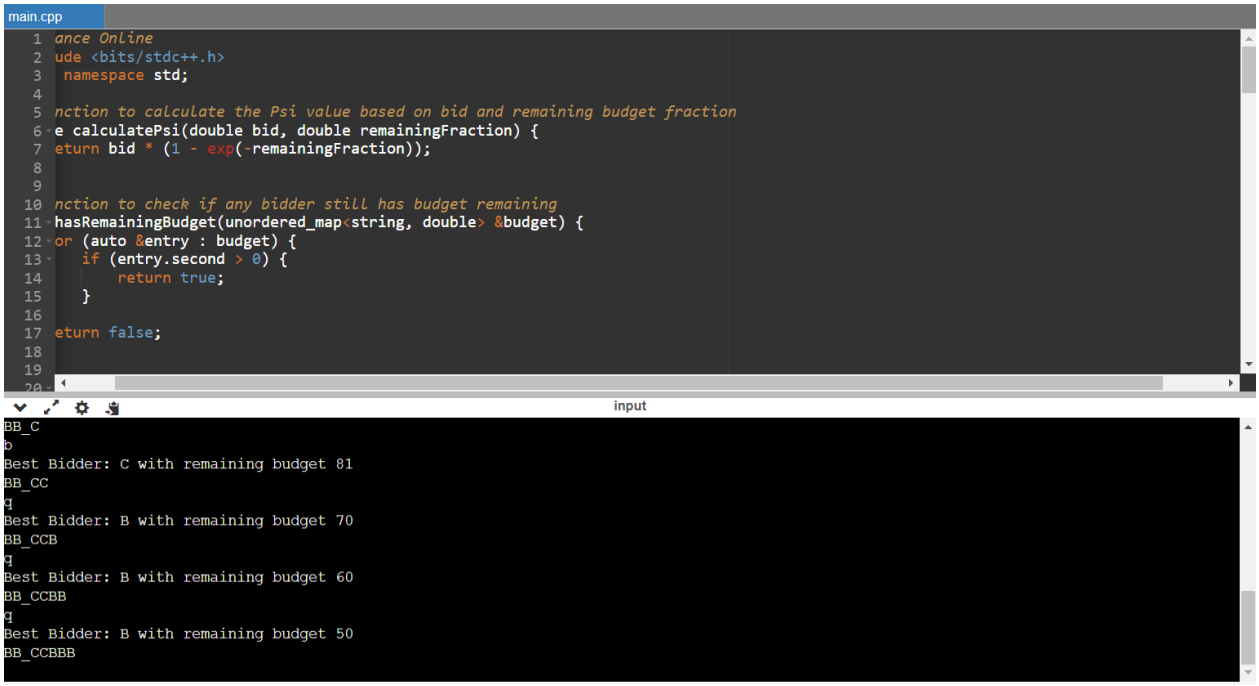
main.cpp

```cpp
//Balance Offline
#include <bits/stdc++.h>
using namespace std;

// Function to calculate the Psi value based on bid and remaining budget fraction
double calculatePsi(double bid, double remainingFraction) {
    return bid * (1 - exp(-remainingFraction));
}

// Function to check if any bidder still has budget remaining
bool hasRemainingBudget(unordered_map<string, double> &budget) {
    for (auto &entry : budget) {
        if (entry.second > 0) {
            return true;
        }
    }
    return false;
}
```

input

```
Best Bidder: A with remaining budget 100
Allocated so far: A
Best Bidder: A with remaining budget 90
Allocated so far: AA
Best Bidder: A with remaining budget 80
Allocated so far: AAA
Best Bidder: A with remaining budget 70
Allocated so far: AAAA
Best Bidder: A with remaining budget 60
Allocated so far: AAAAA
Best Bidder: A with remaining budget 50
Allocated so far: AAAAAA
Best Bidder: A with remaining budget 40
Allocated so far: AAAAAAA
```

Finally, to conclude our discussion on the Ad-Words problem, we went a step ahead and implemented an **ONLINE** version of the **Generalized Balance Algorithm**.



BALANCE ALGORITHM	GENERALIZED BALANCE ALGORITHM
• The Balance Algorithm is a specific type of algorithm designed to distribute tasks or resources **evenly** across multiple entities (e.g., servers, processors, advertisers) to achieve a balanced load. • Due to its simplicity, the Balance Algorithm can be very efficient and quick to implement. • It's suitable for scenarios where a fast, basic distribution is needed without considering too many variables. **Example:** Even distribution of jobs among a set number of processors.	• Extends the basic principles of the Balance Algorithm to accommodate a **broader range of factors** and more complex criteria for distribution. • It aims to optimize the distribution process by considering multiple objectives, such as remaining budget of the advertiser, amount spent so far, or balancing load according to the bid for the query. **Example:** Resource allocation in advertising networks, where bids, budget constraints, and advertiser priorities are considered.

REFERENCES:

MINING MASSIVE DATASETS, SECTION 8.4

HTTP://INFOLAB.STANFORD.EDU/~ULLMAN/MMDS/CH8.PDF

ADWORDS AND GENERALIZED ON-LINE MATCHING

HTTPS://EECS.HARVARD.EDU/CS286R/COURSES/SPRING05/ONLINEMATCH.PDF
HTTPS://WEB.STANFORD.EDU/~SABERI/ADWORDS.PDF

ONLINE MATCHING AND AD ALLOCATION

HTTPS://WWW.CS.CMU.EDU/~ARIELPRO/15896S15/DOCS/PAPER13B.PDF

LEARNING IN BIG DATA-ANALYTICS [LECTURE 4]

HTTPS://GDS.TECHFAK.UNI-BIELEFELD.DE/_MEDIA/TEACHING/2021WINTER/LBDA/LECTURE-BALANCEADWORDS-081220.PDF

LECTURE SLIDES FROM SEMESTER 6 – MINING OF MASSIVE DATASETS

GOOGLE CLASSROOM

PARALLEL PROGRAMMING

HTTPS://LEARN.MICROSOFT.COM/EN-US/CPP/PARALLEL/CONCRT/HOW-TO-PERFORM-MAP-AND-REDUCE-OPERATIONS-IN-PARALLEL?VIEW=MSVC-170

APACHE HADOOP MAP/REDUCE

HTTPS://HADOOP.APACHE.ORG/DOCS/CURRENT/HADOOP-MAPREDUCE-CLIENT/HADOOP-MAPREDUCE-CLIENT-CORE/MAPREDUCETUTORIAL.HTML

APACHE SPARK MAP/REDUCE

HTTPS://SPARK.APACHE.ORG/DOCS/LATEST/API/PYTHON/GETTING_STARTED/INDEX.HTML