Problem set 4, Part 1, Theory

Problem 1, Memory and Caching

a) For each of the three types of cache misses, describe it, and ways of avoiding such misses

Three types of cache misses:

- **Cache conflict**

  Miss occurring because several memory blocks are mapped to the same area of the cache, and some data will be discarded to solve the conflict.

- **Cache capacity**

  Miss occurring because the program needs to reference a collection of memory blocks of a size which exceeds the capacity of the cache memory, and some of the data will be discarded from the cache.

- **Cache compulsory**

  Miss occurring when the program references the first block of memory from an empty cache memory, for example after a start up. The program has to access other, slower, memory areas like the RAM to find the referenced data.

Ways to avoid misses due to:

- Conflict

  Introducing associativity will reorder how the blocks are arranged and may decrease number of conflicts

- Capacity

  Increase the cache capacity to match the required space for running necessary programs.

- Compulsory

  Increasing the block size in the cache

b) For each of the following code snippets, determine if the access pattern exhibits spatial locality, temporal locality, both, or neither and explain why

I) The for - loop loops through neighboring memory addresses one by one. This access pattern will exhibit **spatial locality**.

II) The double for-loop will loop through neighboring memory addresses multiple times in a strict pattern. This access pattern will exhibit **spatial locality** as I), but also **temporal locality** as it access the same address a hundred times with new for-loop (compared to I)).

III) The double for-loop indexes the arrays by multiplying the counter j by a constant, here 1000, 2000 and 3000. This will lead to the program accessing areas of the array which are not neighboring memory addresses. The program will still as in II) access the same addresses multiple times which leaves the access pattern exhibiting **temporal locality**.

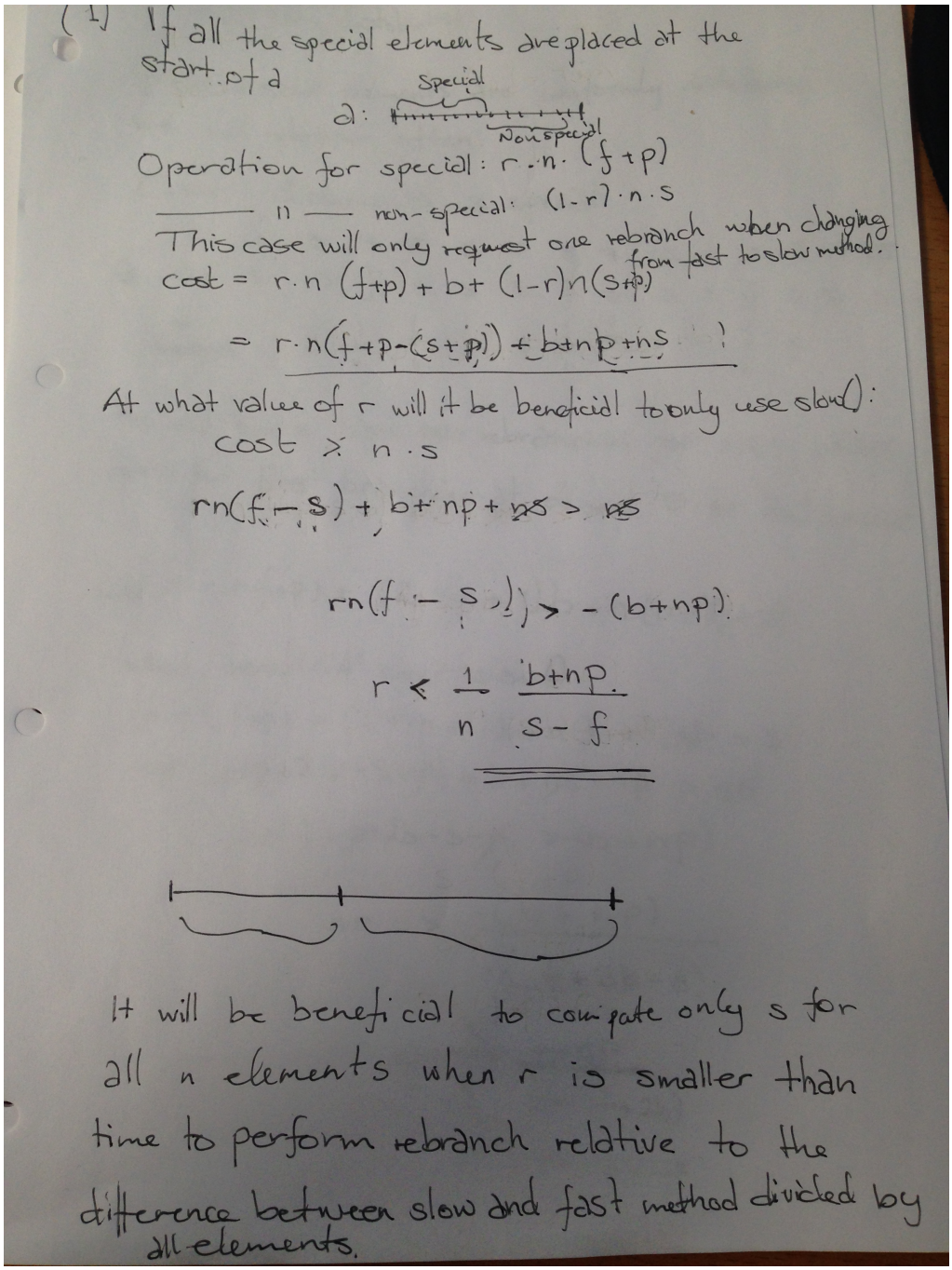
Problem 2, Branching

a) What is branch prediction? How can it improve performance?

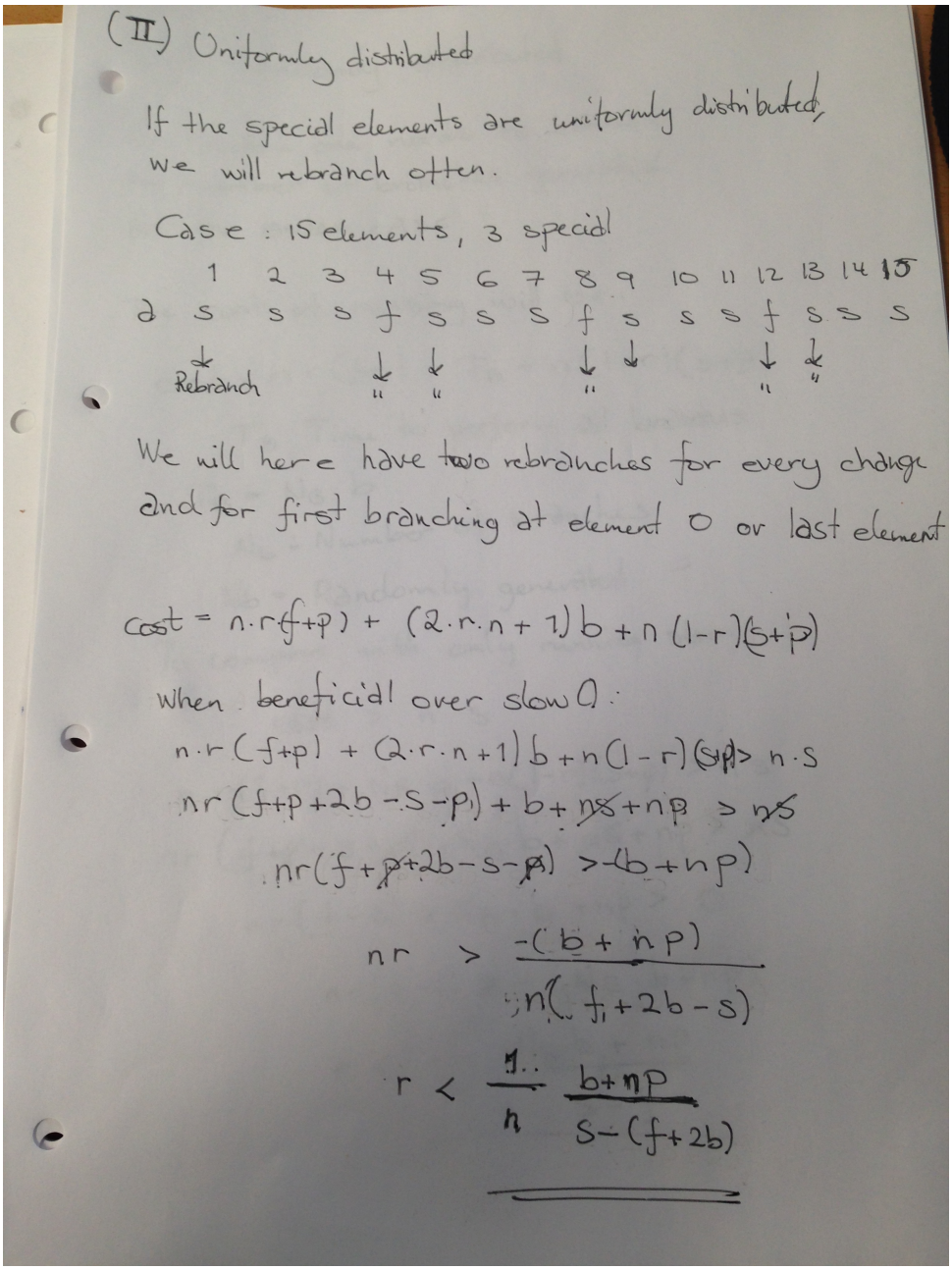
Branches in computers will experience to come to a crossroad where it can continue in two different directions. If the branch chooses one of the directions, and later it proves to be the wrong direction. The branch will have to start over again in the other direction. Unfortunately the branches often don't which direction to choose when it approaches the crossroad. One way to solve this issue is to build branch predictors which predicts the direction of the branch. If the predictor is right the system will experience improved performance, and no stop and start of branches in the middle of an instruction. If the predictor is wrong the system performance will decrease as the branch will need to restart. If one develops good branch predictors the system will improve it performance.

b) Consider the following code, at value of r will it beneficial to remove the branch, and use slow() for all the elements?

Case (I)



Case (II)



(II) Uniformly distributed

If the special elements are uniformly distributed, we will rebranch often.

Case: 15 elements, 3 special

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s	s	s	f	s	s	s	f	s	s	s	f	s	s	s
↓			↓	↓			↓	↓			↓	↓		
Rebranch			"	"			"				"	"		

We will here have two rebranches for every change and for first branching at element 0 or last element

$$\text{cost} = n \cdot r(f+p) + (2 \cdot r \cdot n + 1)b + n(1-r)(s+p)$$

When beneficial over slow 0:

$$n \cdot r(f+p) + (2 \cdot r \cdot n + 1)b + n(1-r)(s+p) > n \cdot s$$

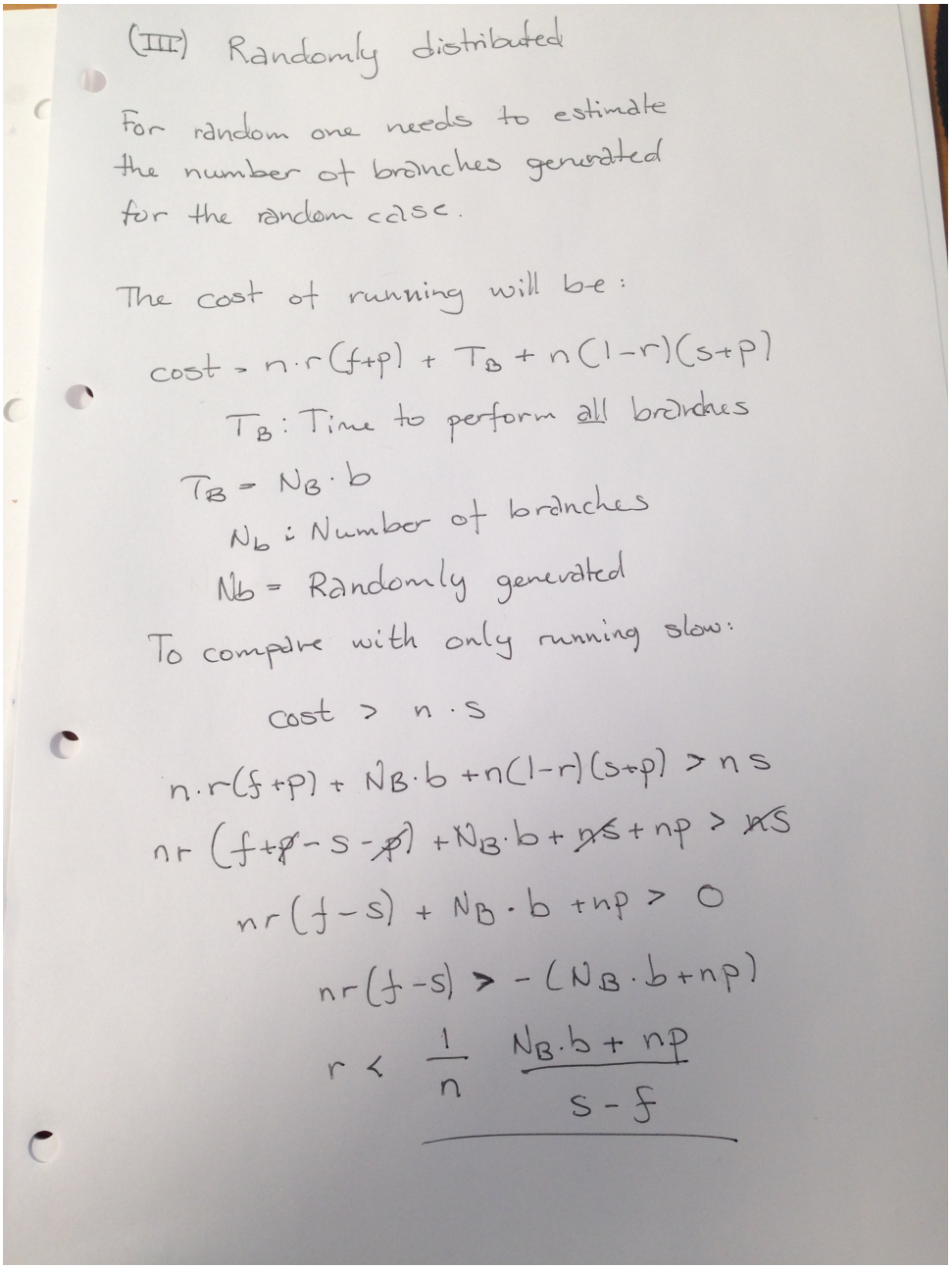
$$nr(f+p+2b-s-p) + b + ns + np > ns$$

$$nr(f+p+2b-s-p) > -(b+np)$$

$$nr > \frac{-(b+np)}{n(f+2b-s)}$$

$$r < \frac{1}{n} \cdot \frac{b+np}{s-(f+2b)}$$

Case (III)



(III) Randomly distributed

For random one needs to estimate the number of branches generated for the random case.

The cost of running will be:

$$cost = n \cdot r(f+p) + T_B + n(1-r)(s+p)$$

T_B: Time to perform all branches

$$T_B = N_B \cdot b$$

N_b: Number of branches

N_b = Randomly generated

To compare with only running slow:

$$cost > n \cdot s$$

$$n \cdot r(f+p) + N_B \cdot b + n(1-r)(s+p) > ns$$

$$nr(f+p-s-p) + N_B \cdot b + ns + np > ns$$

$$nr(f-s) + N_B \cdot b + np > 0$$

$$nr(f-s) > -(N_B \cdot b + np)$$

$$r < \frac{1}{n} \frac{N_B \cdot b + np}{s-f}$$

Now we have the expression for r, but we still need to find a measure for N_B. One way to find the value is to simulate the random distribution of special elements for various number of r and placement of elements. A such simulation has been performed in Matlab, and gives the results shown on the next pages.

Table of Contents


```
%
% Purpose: Simulate branching for array containing randomly
 distributed
% special cases
%
% Made by:
%       Even Florenes NTNU 2016
%
```

Initalize simulations

```
% Set number of full simulations
nSimulations = 20;

% Set number of repetions
nRepetions = 100;

% Set length of array
N = 100;

% Set fixed ratio of special cases
rArray = 0:1/N:0.5;
rebranchesavgR =zeros(1,length(rArray));
rebranchesTotal = zeros(nSimulations,length(rArray));
% Initalize results array
rebranchesSimulation = zeros(1,nRepetions);
```

Simulate

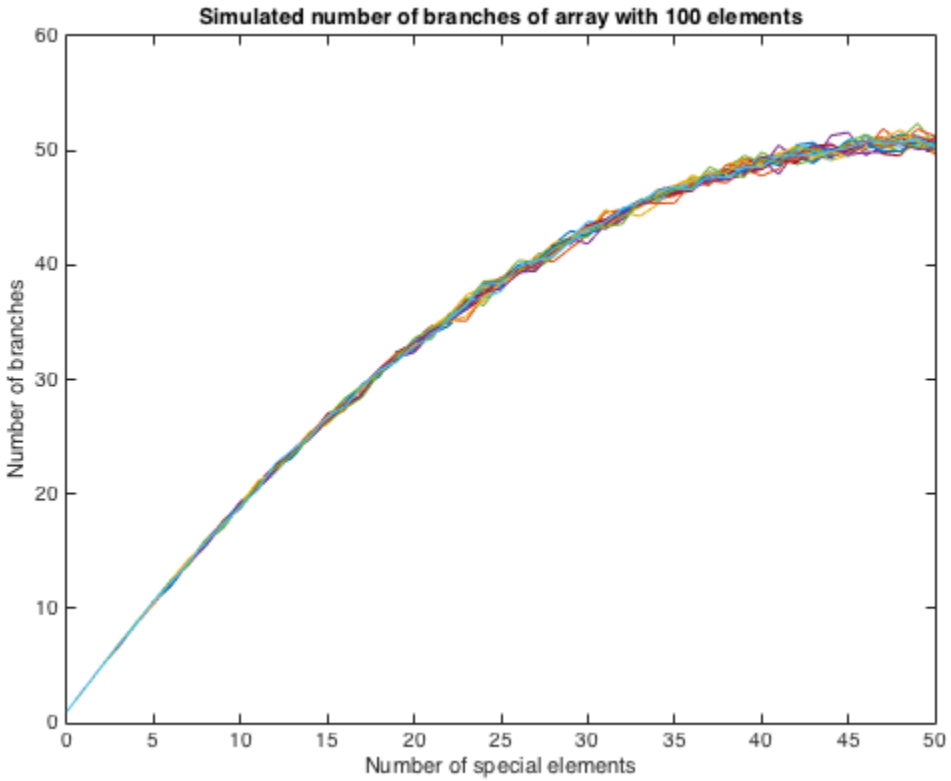
```
for i = 1:nSimulations

    for j = 1:length(rArray)

        r = rArray(j);

        for k = 1:nRepetions

            % Compute number of special cases
            nSpecial = round(r * N);

            % Create temporary array with all array positions
            aIndex = 1:N;
```

```
            % Initalize special positions array
            specialPositions = zeros(1,nSpecial);


            for l = 1:nSpecial
                randomInd = round ( 1 + rand * ( N - l + 1 - 1) );
                specialPositions(l) = aIndex( randomInd );
                aIndex( randomInd ) = [];
            end

            % Initalize simulation array
            a = zeros(1,N);
            a(specialPositions) = 1 ;

            % Counter for number of branches
            rebranches = 0;

            % Display to screen simulation status
            %disp(['Simulate case :', num2str(k)]);

            for l = 1:N

                if l == 1
                    if ( a(l) == 0 )
                        rebranches = rebranches + 1;
                    end
                else
                    if ~isequal(a(l-1),a(l))
                        rebranches = rebranches + 1;
                    end
                end % if j

            end % for j

            rebranchesSimulation(k) = rebranches;

        end % for k
        rebranchesavgR(j) = sum(rebranchesSimulation)/(N);
        %disp(['Average number of rebranches: ',
 num2str(sum(rebranchesSimulation)/N) ]);


    end % for j

    rebranchesTotal(i,:) = rebranchesavgR(:);


end % for i
```

Display result to screen

```
plot( N* rArray, rebranchesTotal),xlabel('Number of special
 elements'),ylabel('Number of branches');
```

```
title('Simulated number of branches of array with 100 elements')
```



Published with MATLAB® R2015a

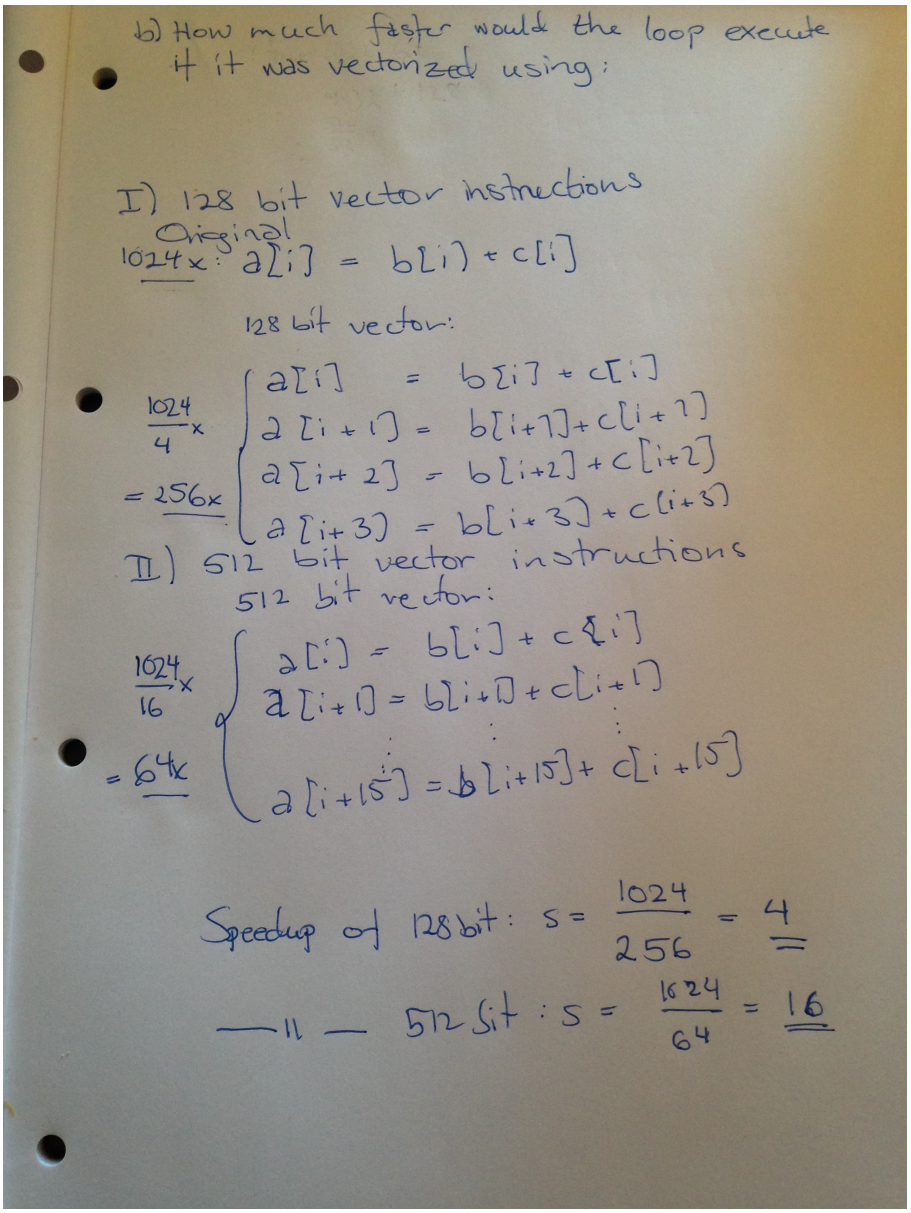
Results from Matlab gives a good approximation of the value for N_B for different number of special elements. The expression for r and approximation of N_B gives an equation for the boundary between using predictor or only using slow algorithm.

Problem 3, Vectorization

a) What are SIMD instructions (on x86 processors) ?

SIMD is short for single instruction multiple data which is one implementation of parallel instruction set. SIMD can vectorize scalar code by performing a single instruction for multiple times in parallel. SIMD can not be used to perform different instructions in parallel as Single Instruction in SIMD suggest. SIMD can be used to speed up for loops performing the same operation every loop but for different data.

b) How much faster would the loop execute if it was vectorized



Problem 4, Optimization

Speedup alternative 1:

In the current implementation the power of x for some n is calculated every iteration.
For example:

i = 0

r += pow(x, 0*2 + 1);

i = 1

r -= pow(x, 1*2 + 1);

As i increases towards 100 this operation will be costly for performance. One alternative implementation given possible speed up would to keep the calculated pow(x, i*2 +1) for every iteration and only multiple with x^2 for every iteration:

Implementation:

```
double slow_sin_speedup(double x) {
	double r=0; int i;
	double x_power = x;
	for(i=0; i<100;i++) {
		if( i%2 == 0){
			r += x_power / factorial(i*2 + 1);
		}
		else {
			r -= x_power / factorial(i*2 + 1);
		}
		x_power = x_power * pow(x,2);
	}
	return r;
}
```

Speedup alternative 2:

One alternative method for speeding the code snippet would be to build a branch predictor. The condition for the if- else statement, i modulo of 2, will be true every other time. The condition will be true for *i* which is even, and false for *i* odd. The behavior of the if-else case is well-known, and branch predictor can predict true for the first case in the iteration (i = 0) and the false for the next iteration (i = 1), and the same pattern continues to a i = 100.

Speedup alternative 3:

One way of speeding up the code would be to calculate factorial for i = 1 to i = 100 prior to calling the function, and store all the values in an array. This is a method which is not very elegant, but for this case where there are “only” 100 iterations, it will give great performance enhancement.

Speedup alternative 4:

Replace functionality of the if-else in the for-loop. The branch choice of if - else can potentially worsen the performance. By introducing a variable *choice,* which is the former condition in the if-else branch, we can remove the if-else and choose operation based on this variable

```
double slow_sin_speedup(double x) {
        double r=0; int i;
        double x_power = x;
        for(i=0; i<100;i++) {
                int choice = ( i%2 );
                r += pow(x * ( 1 % choice ) , i*2+1);
                r -= pow(x * choice , i*2+1);
                r = r/factorial(i*2 + 1);
        }
        return r;
}
```

Problem 5, OpenMP schedules

a)

In static scheduling the iterations are divided evenly between the available threads. In the case of static scheduling we need to make sure that the work load is evenly balanced between the different threads. This implies a static *arg* in the code snippet, to make sure that every thread receives work evenly sized with other threads, leaving no threads idle for a long time waiting for other busy threads.

Completing line 2 for static schedule:

```
int arg = 1024/num_threads;
```

b)

In dynamic scheduling the work load is evenly distributed among threads. Because when one thread is finish performing it's task, it will not go idle but find work somewhere else. Which means every available will stay busy until all work is done. The distribution of work load is performed at run time with a queue system leading to large over head at run time. As the dynamic schedule makes sure to divide the work among available threads we can set the *arg* to a dynamic value. To avoid large over head. The first assigned task to thread should be larger in work than the remaining. As queuing and assigned threads leads to overhead, we should try to minimize the number of times the system needs to assign new tasks.

Completing line 2 for dynamic schedule:

```
int arg = ( 1024 - i ) / num_threads ;
```

c)

Guided scheduling is somewhat a combination of static and dynamic scheduling. Where the threads dynamically grab blocks of iterations, where the blocks are decreasing in size

as the calculation proceeds. With this schedule it would be beneficial that the calculation time increases as *i* increases. Because the sizes of the of the blocks decrease one can increase the work in these blocks.

Completing line 2 for guided schedule:

```
int arg = 1024 / ( ( 1024 - i ) * num_threads );
```